Design-space Exploration of Most-recent-only Communication using Myrinet on SGI ccNUMA Architectures

Rodney Freier Al Davis William Thompson

Dept. of Computer Science
University of Utah
Salt Lake City, UT 84112
{freier,thompson,ald}@cs.utah.edu

UUCS-99-012

December 20, 1999

Abstract

SGI's current ccNUMA multiprocessor architectures offer high scalability and performance without sacrificing the ease of use of simpler SMP systems. Although these systems also provide a standard PCI expansion bus, the bridging between PCI and SGI's ccNUMA architecture invalidates the assumptions typically made by network protocol designers attempting to use Myrinet to reduce communications latencies. We explore the complications introduced by SGI's architecture in the context of designing most-recent-only communications, in which a reader requires only the most recent datum produced by a writer.

1 Introduction

Real-time distributed systems frequently need to communicate updates to specific quatities which change over time. These quantities have been referred to as "signals" [GP94], and they are defined by two important properties. First, the value of a signal is *time-critical* in that only the most recent value matters. Second, setting the value of a signal is an *idempotent* operation; assigning a specific value to a signal multiple times is indistinguishable from assigning that value exactly once. In typical real-time systems, the producer of a signal is often a dedicated process supplying the signal either at a fixed rate or sporadically as dictated by interrupts from measurement hardware. Position information provided by Global Positioning System hardware is a common example of a signal in a real-time system. The consumer of such a signal needs the most recent value as soon as posible after it is generated.

NDDS [GP94] provides the only commercially available example of a most-recent-only network communication API. NDDS uses a publish-subscribe model in which any number of producers may publish a named signal, and any number of consumers can subscribe to these signals. The consumers perceive only the most recent publication. NDDS has provisions for assigning strengths to each producer and decaying the strengths of a publication over time, allowing consumers to perceive the strongest signal. These features allow data produced by redundant backup systems to be used automatically and seamlessly if primary systems fail. NDDS operates over UDP, and has a minimum latency rougly twice that of UDP over 100 megabit

ethernet.

Because latency of communication is a fundamental issue in real-time systems, many designers use special low-latency and high-bandwidth networking hardware and protocols. High performance networking is a rapidly growing field exploring many alternatives in an effort to increase bandwidth and reduce latency. Myrinet [NJB95], GSN [SGI99], FiberChannel, Gigabit Ethernet, and ATM are examples of current network technology, differing in performance and provisions for reliable transmission, flow control, and quality-of-service guarantees. Systems such as SCRAMnet [TB98] and other replicated or distributed shared memory systems also provide low latency communication between heterogenous systems and eliminate the protocol layer necessary with networks.

We have chosen Myrinet because of its low cost/performance ratio and its use of standard PCI interfaces. Myrinet also has particular appeal to protocol designers because of its flexibility. Each Myrinet PCI network interface card is equipped with a processor and can be programmed to do protocol-specific processing. Thus, many protocol varieties can easily be implemented over Myrinet. BIP [PTW98], GM, U-Net [TE95] are examples of packet stream protocols using Myrinet. VMMC [CD97], Hamlyn [GB95], Direct Deposit [MRS95], and Direct Access U-Net provide examples of sender-based protocols, in which the sender specifies an address in receiver process memory where a transmitted packet is to be placed. All of these systems have the goal of reducing the latency of communication as much as possible.

This protocol flexibility, along with its high bandwith and low latency, have also made Myrinet popular with real-time systems engineers. Some manufacturers of single-board computers intended for real-time and embedded use have integrated Myrinet into their board designs.

Because our real-time work focuses on producing interactive virtual environments, the SGI Onyx2 architecture has proven indispensable for its high-performance rendering engines and parallel-processing facilities. However, this architecture has proven somewhat unfriendly to those using PCI devices. The goal of this work is to explore the design space of implementing most-recent-only communications protocols using Myrinet on current SGI architectures.

The paper is organized as follows. Section 2 introduces the algorithms that we use for most-recent-only communication and describes the desired features. Section 3 describes Myrinet and its performance in Onyx2 systems. Section 4 explores several design alternatives. Section 5 adds additional functionality, explores the new implementation alternatives, and gives performance results.

2 Most-recent-only Communication

We desire a communications mechanism that allows a writer to communicate a fixed-sized data structure (the signal) to a reader in such a way that the reader only reads the most recent value written. We first assume shared memory is available, to simplify the presentation, although our goal is to use a network for communication.

The simplest approach is to store the signal in one semaphore-protected buffer shared by the partners. Each one acquires the semaphore before modifying or examining the signal. While simple, this approach has a significant drawback: Neither partner can access the signal without blocking progress by the other. Therefore, each must minimize the time spent reading or writing the signal. At best, each partner could copy the data to perform reading or writing. However, this copy can introduce more latency.

A second approach is to use multiple buffers for the signal. Although double-buffering might seem to provide a solution, this scheme still allows either wasted effort by the writer or contention between the partners. Triple-buffering overcomes these problems, allowing concurrent reading and writing [GLP83]. This technique is commonly used in PC video cards to allow a graphics application, the "writer" of a frame of graphics, to proceed independently of the monitor, which acts as the "reader" of a frame of graphics and must consume (i.e. display) these frames at a fixed rate [PH99, DF96].

```
Signal Buffer[3];

int In  = 1;
int Out = 0;
int LR  = 0;

Write(Signal value)
 {
   Buffer[In] = value;
   LR = In;
   In++; if (In >= 3) In = 0;
   if (In == Out) {
      In++;
      if (In >= 3) In = 0;
   }
 }

Signal *Read()
 {
   Out = LR;
   return &Buffer[Out];
 }
```

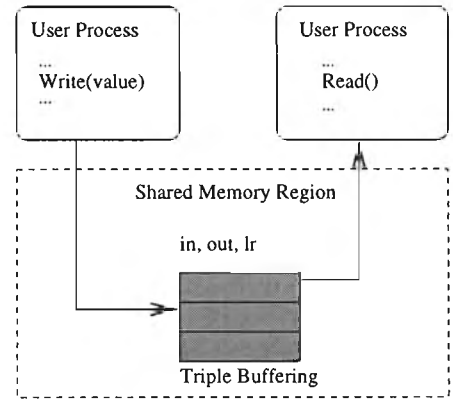


Figure 1: C-like Pseudo code for communicating a signal.

Figure 1 gives pseudocode for communicating a signal using triple-buffering in shared memory. Note that it uses no semaphoring primitives and allows both the reader and writer to proceed at any time without blocking. In essence, triple-buffering provides one buffer for the reader to examine, one to contain the next most recent data produced by the writer, and another which can receive new data. Atomicity of the read operation is ensured by accessing the LR index with an integer operation.

This approach is simple and efficient when shared memory is available. No unnecessary copies of the data are made, neither process need ever block, and latency is minimized. However, we desire these same properties for communication between processes on physically distant hosts connected by a network.

The most-recent-only paradigm, when implemented as a network protocol, allows two convenient optimizations. First, whereas byte- or packet-stream protocols require queues to handle the data stream, a most-recent only protocol can replace these queues with triple-bufferring and avoid blocking due to a queue that is empty (readers) or full (writers). Second, if the underlying network is unreliable, guaranteed transmission can be easily imlemented, since a most-recent-only protocol requires an acknowledgement for only the most recently transmitted data.

By applying this algorithm twice and using separate threads to handle network transmission and reception, we arrive at a simple design for most-recent-only communication over any network. We will continue to refer to the communicated data as a signal, and refer to this design as a signal notification engine (SNE). Figure 2 illustrates the design of a generic SNE.

The triple-buffering on each side prevents the user threads from blocking for any network-related reason, such as loss of connection with the partner or blocking of the underlying network protocol operations. The network threads handle all connection establishment, transmission, reception, and reliability issues. The

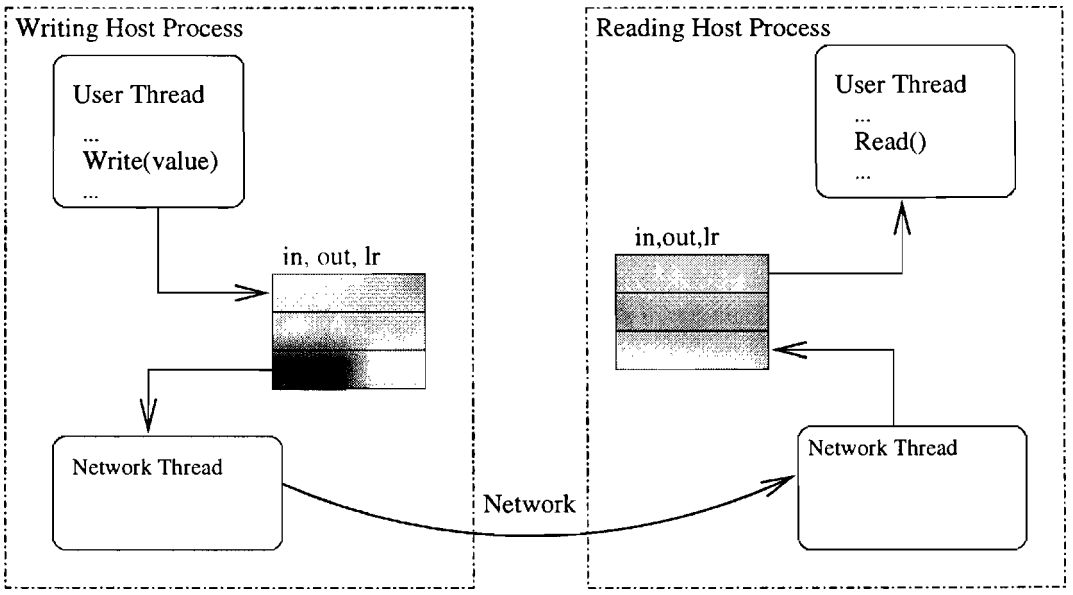


Figure 2: Generic Signal Notification Engine

user processes can continue to call the *read* or *write* functions even if a partner is not actually present. When one appears, the system can easily reinitialize and continue. There is no need for the writer to know if a partner is present, and to the reader, having no partner is equivalent to receiving no new data.

The simple interface between threads also allows the network thread to select the most appropriate method of communicating with the partner. This thread can choose shared memory, TCP, UDP, or other available communications mechanmisms based on the location of the partner process.

3 Myrinet

As discussed, our target network is Myrinet, a high-performance local area network supplied by Myricom. A single Myrinet link provides 1.28 Gbps full duplex bandwidth and an extremely low bit error probablility of 10^{-15}. A network is formed by connecting links to multi-port switches. Packets are source-routed through this network; i.e. a sender appends a set of routing bytes to the head of a packet which are stripped off by a switch one at a time to determine the correct output port for that packet. The latency across a switch is about about 0.1 μs

The hardware itself ensures only in-order delivery; it does not ensure reliable delivery. However, both switches and network interface cards perform CRC generation and checking, and Myrinet will drop a packet only under very clear circumstances, which can often be avoided by design.

The network interface cards (NICs) are fully programmable, providing a CPU (the LANai 4.1), SRAM, and three DMA (Direct Memory Access) engines for transferring data between the host and the NIC and between the NIC and the network. A complete transmission of a packet over Myrinet consists of four stages: a transfer from the sending host to the NIC's SRAM, from NIC's SRAM to the network, from the network to the receiving NIC's SRAM, and from the receiving NIC's SRAM to the receiving host's memory. The memory bus on the NIC is clocked at twice the speed of the LANai, which allows multiple DMA engines and the processor to work simultaneously, so the four transmission stages can proceed simultaneously. [1]

[1]These transfers are not considered copies, since they are necessary for any data to be transmitted over the network. Only transfers from host memory to host memory by the host processor, or from NIC memory to NIC memory by the NIC processor, are considered undesireable copies.

The SRAM acts as synchronous memory on the PCI bus, supporting shared-memory interaction between the host and NIC CPUs.

These features make it easy to implement protocols which perform much of the critical processing directly in the network interface hardware, and which allow user processes direct access to the network. This avoids the latency overhead incurred by using the OS to mediate all network activity, and interprocess protection can still be maintained.

Various protocols can thus be used with Myrinet by installing the appropriate Myrinet control program (MCP) in the NIC and using the associated drivers and user-level libraries. Fast Messages [SP97], Hamlyn [GB95], BIP [PTW98], Trapeze [JC99], and Myricom's GM are examples of available protocols.

Although faster Myrinet interface cards are now available, all performance measurements mentioned here are specific to 32-bit PCI LANai 4.1 NICs.

3.1 Myrinet Performance on Onyx2

Current generation SGI architectures, such as the Onyx2 and Origin 200/2000 systems, share the same fundamental design. Though the results here were obtained on a four-processor R12000 Onyx2, they are typical of any current SGI platform. Figure 3 shows the raw bandwidth and latency as a function of DMA transaction size for the Onyx2. These figures agree with those obtained from running Myricom's `hswap` benchmark. The DMA performance is far from the 133 MB/s maximum, which is nearly reached by good PC PCI implementations. Note specifically that the cache line size is visible in these graphs. The SGI PCI architecture allows only one cache line to be transferred at a time from a PCI device. This transfer is followed by a mandatory wait time to allow other PCI devices to proceed and to allow the XIO internal fabric to handle cache coherency.

Myricom's `hswap` benchmark determines that programmed I/O (PIO, i.e, accessing the memory directly using host processor read or write instructions) to the interface card achieves a maximum bandwidth of 38 MB/s on SGI systems. A test of the latency incurred in a simple PIO handshake with the Myrinet NIC reveals a typical handshake time of 3.7 μs. One handshake sequence proceeds as follows: 1) the host processor sets a 32-bit memory location in NIC SRAM, and begins spinning to detect a reset of this location; 2) the LANai processor, programmed to spin in a loop waiting for a change in this location, detects the change and resets the location; 3) the host processor detects the reset. One handshake therefore consists of one PIO write by the host and at least one read. This result is higher than what we would expect for PCI PC's,

Latency results for the MyriAPI also reveal deficiencies of the PCI implementation. We ported this API to IRIX 6.4, and testing revealed a minimum latency of 105 μs for a 4-byte packet. A similar port was completed by [MG97], and obtained comparable results. Other platforms achieve between 50 μs and 70 μs with the same API and NIC software.

3.2 PCI Irregularities

The bridging between a PCI bus and an internal system bus is usually expected to provide certain guarantees. For example, it may be expected to maintain the ordering of writes and flush all writes before a read is allowed.

Many protocols for Myrinet have relied on these properties to eliminate the host OS from involvement in the critical path of packet transfer. If the PCI system provides ordering guarantees, a user processes can interact directly with the network interface card through shared data structures in NIC SRAM, just as two host processes can communicate using shared memory. This avoids the use of the OS as a mediator of all network activity. Since the OS can be a source of unpredictable delays, this measure is often used to reduce latency. This is the approach taken by U-Net [TE95].

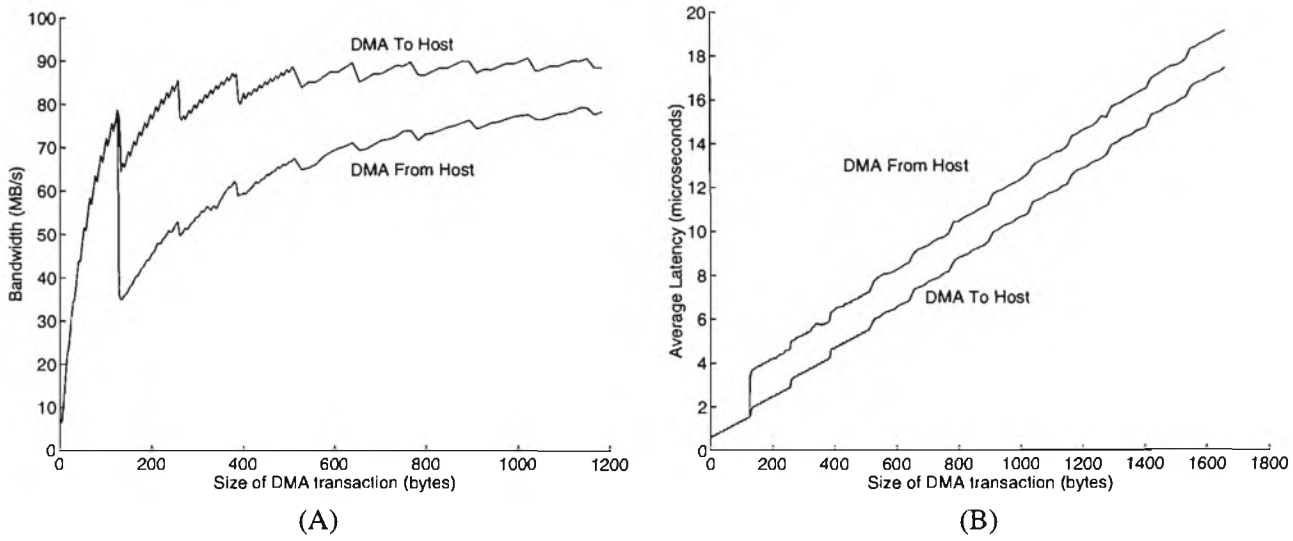


Figure 3: PCI DMA Bandwidth (A) and Latency (B) on the SGI Onyx2

A critical part of the interaction between a user process and the NIC is determining when a DMA operation is complete, so that the user process can safely examine the transmitted data. However, the bridge between SGI's internal XIO bus and the PCI peripheral bus relaxes the DMA ordering guarantees, thus making this determination less straightforward. Two DMA operations targetting separate cache lines of host memory may actually commit to memory out of order, thus making it impossible to signal the completion of a data DMA with a control DMA. Further, a PIO read of a PCI device can return its results even though several DMA writes to host memory from that device have been initiated but not yet completed. This makes it impossible for a host program to read an NIC register to determine if a DMA operation is complete. The only guarantee involving DMA to host memory offered on SGI systems is that an interrupt routine for a PCI device will not be called until all pending DMA operations from that device have committed to host memory. Of course, PIO operations must resolve in order; otherwise there would be no way to reliably control the PCI device. DMA operations from host memory to NIC SRAM resolve in order.

These facts leave us three alternatives to allow a user process to determine completion of a data transfer. First, we could use PIO exclusively to transfer data between a host process and the NIC. The maximum bandwith obtainable would be severely reduced, but since PIO operations are strictly ordered, no ambiguity would result on SGI systems. Second, we could arrange to place signal bits inside DMA transactions. We could then detect completion of a DMA operation by waiting for the signal bits to contain the correct values. Unfortunately, since individual cache lines of a single DMA operation can resolve out of order, this would require at least one signal bit in each cache line of data, which seems unwieldy. Finally, we can simply accept the need for interrupts and use them to signal data transfer completion, even though this brings the operating system back into the critical path.

4 Signal Notification Engine Designs using Myrinet

Section 2 introduced a simple design for a generic signal notification engine. This design could use Myrinet, with some choice of the available protocol software as the underlying communications layer. However, with Myrinet another simple optimization becomes possible: the network interface card can perform the processing done by the network threads.

Any control program running on myrinet must provide enough functionality to send or receive packets and interact with the host. It is not a significant additional burden for this control program to receive into a specific buffer as dictated by the triple-buffering variables In, Out, and LR. This small modification allows us to eliminate the threading necessary with the generic SNE.

Latency is reduced by this approach in three ways. First, the complexity of the Myrinet control program is actually reduced. Instead of managing send and receive queues, the control program need only manage three buffers for a given connection. Second, the user process is not involved at all in facilitating packet reception. Packets can be placed in their final destination by the Myrinet control program without any help or information from the host program. Finally, reliability is more easily implemented when only the last received packet is important. Therefore, it is very likely that this approach will have lower latency than a well-tuned packet-stream protocol using the same host and network hardware.

However, such an approach leaves us with a special purpose network. Other protocols for Myrinet are general purpose and support the construction of higher-level protocols on top of them. It would be difficult, for example, to design a packet stream protocol which used this SNE design as an underlying layer. Further, desiging Myrinet control programs is difficult, and time may be better spent utilizing established protocols.

We describe three approaches to implementing a Myrinet SNE directly on the Myrinet hardware.

4.1 Using PIO

Figure 4 illustrates our first SNE design which uses the Myrinet hardware directly. Here, we avoid DMA ordering issues by using PIO exclusively. The In, Out, and LR variables and the triple-buffers are kept

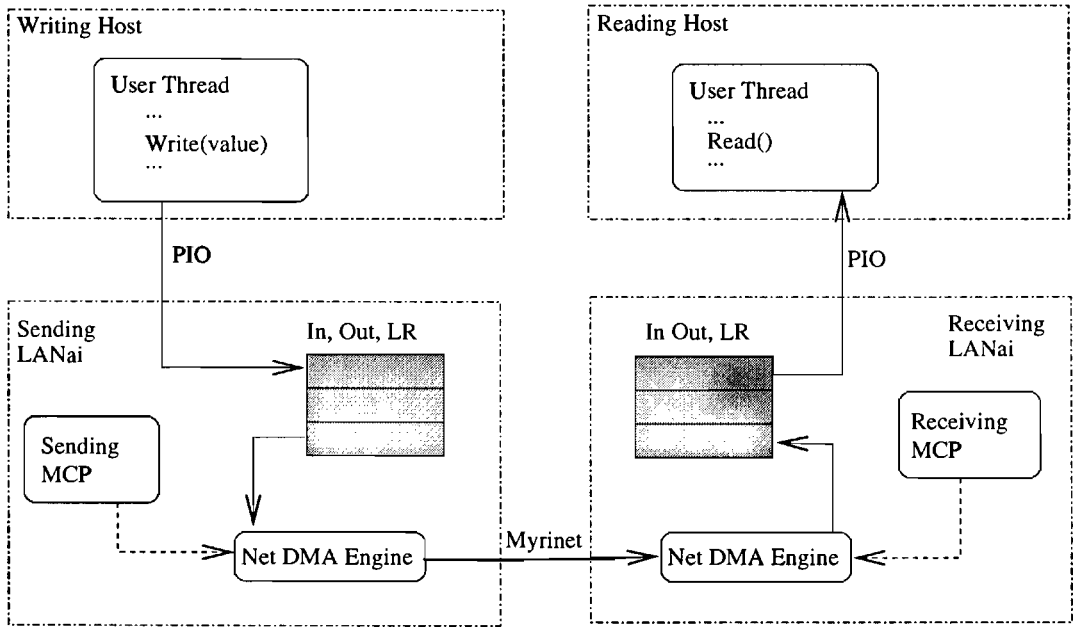


Figure 4: Myrinet SNE design using PIO.

in LANai SRAM. The triple-buffering algorithm is used twice, once between the writing host program and

the sending MCP, and once between the receiving MCP and the reading host program. This allows the host programs to proceed independently of any network-related delays at either end of the communication.

The communications process illustrated in figure 4 proceeds as follows. The writing user thread calls the *write* function, which copies the given value into the buffer with index In. As in the triple-buffering algorithm previously described, LR is set equal to In, and then In is incremented and checked against Out. Since these variables are kept in the NIC SRAM, these accesses are all via PIO.

The sending MCP repeatedly checks to see if Out equals LR as a part of its work loop, and if this equality is not true when the Net DMA engine is available for sending, it sets Out equal to LR and uses the network DMA engine to send the data, with a prepended header and appended checksum, to the partner. Since many connections may be active between the receiving host and others, the header contains connection identificaiton information. It also contains a sequence number so the packet can be positively acknowledged. The MCP makes note of the time when this packet was sent, and if an acknowlegement is not received within a specific time, the packet is resent.

On the receiving side, the Net DMA engine for receiving packets is always active, and alerts the receiving MCP when a packet header has been received. The MCP must examine the header to determine which connection is addressed. The MCP can then instruct the net DMA engine to receive the remainder of the packet (the signal data) into the correct buffer, which is the one indexed by In. The MCP must verify the checksum once the packet has been completely received. If the checksum indicates corruption, the reception can be ignored. Note that by virtue of the triple-buffering scheme, we can be certain that the buffer indexed by In contained old data, and writing over it with a corrupt packet has caused no damage. It can be cleanly ignored simply by not setting LR equal to In. If the checksum is valid, as before LR is made equal to In, In is incremented and checked against Out. A positive acknowlegement packet is sent back to the writer.

The reading host thread calls *read* whenever new data is desired. This function simpy compares Out with LR. If they are not equal, new data is available at index LR, and Out is made equal to LR. After this, the user thread can examine the data indexed by Out using PIO for any length of time; it is guaranteed to be a static copy of the most recent signal available at the time of the *read* call.

While the use of PIO severely restricts the bandwidth, this design is simple and will work with any PCI architecture, including SGI's.

4.2 Using DMA

Using DMA for transferring data will improve the bandwith of this communication system and reduce the latency for communicating large signals, though the overhead of using DMA operations will increase the latency for small signals. While the SGI architecture is an important target, we present a DMA scheme that would function correctly on more standard PCI systems and that can be easily modified to work on SGI systems.

Figure 5 shows a design which uses DMA for transfers to and from the host. Note that the writer's side has become more complicated. Since the host DMA engine and the sending net DMA engine can act independently, it is prudent to separate them with another application of triple-buffering. Thus there are two sets of buffers and index variables. One of the sets of buffers is in host memory, so that the host processor can set these values without performing IO operations. The index variables for these buffers are kept in NIC SRAM, so that they can be shared with the sending MCP. This arrangement allows the user thread, the host DMA engine, and the net dma engine to proceed independently. Note that on this end, there is no possibility of DMA reordering.

The reader's side has also become more complex. First, note that a ring buffer has been added to accept incoming packets. In the previous scheme, the final destination of the data in any packet was a buffer in NIC SRAM. We could guarantee that one would be available and that the DMA engine could immediately fill that buffer. With this scheme, the final destination is in host memory, and since the host DMA engine

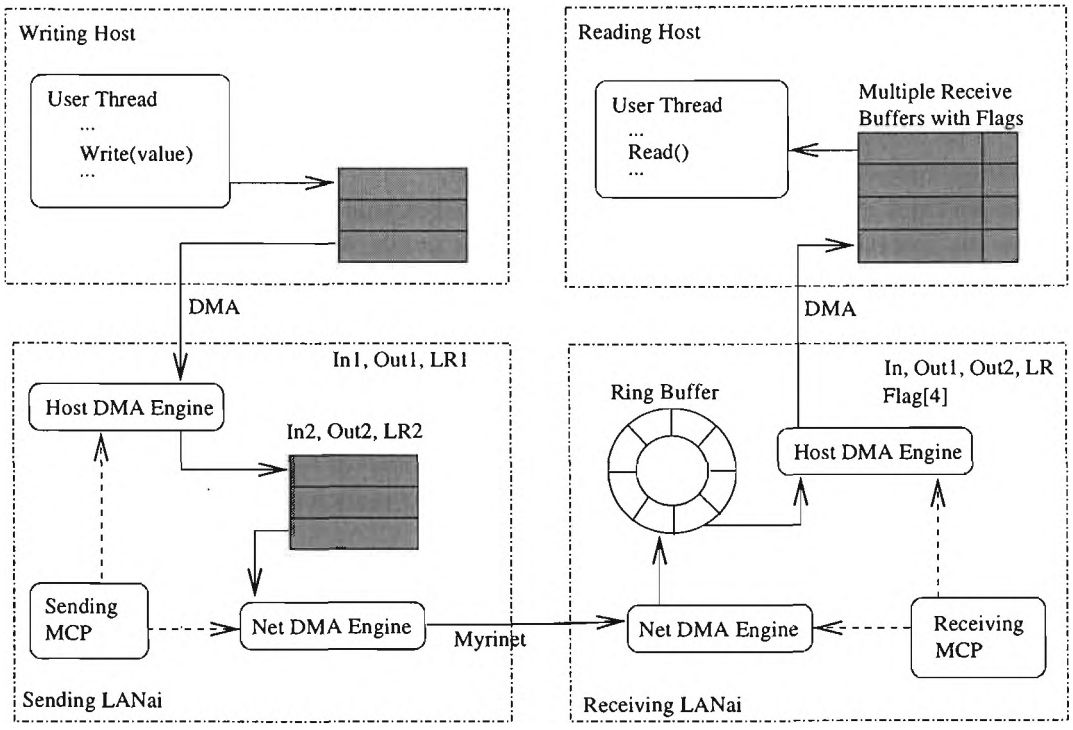


Figure 5: Myrinet SNE design using PIO.

operates independently and may be busy at any given time, we must have a temporary holding place for packets. Therefore, a simple ring buffer is used.

Note that this introduces a problem not present with the PIO version: the ring buffer may fill. If it is full when a packet comes in, the safest thing for the receiving MCP to do is to drop the packet. If it halts the net DMA engine and tries to wait for the ring buffer to empty enough to hold the packet, a timeout may occurr in the Myrinet hardware, and the packet would be dropped anyway. With the previous version, there was never a reason to drop an incomming packet, since buffer space was always available for any valid packet. However, this problem is not serious. The acknowlegement and retransmission scheme used in the PIO version can also be used here, and any dropped packets will be retransmitted. Further, this approach allows an entire packet, header and body, to be received from the network in one operation. In the previous version, only the header was received. Once it was checked, the rest of the packet was received into the correct buffer. Using only one DMA opertion for the whole packet halves the DMA overhead.

The MCP extracts packets from the ring buffer, examines their headers and the indexing variables to determine the correct host buffers, and uses the host DMA engine to transfer them to host memory. Note that on most PCI systems, the usual triple-buffering algorithm, with the indexing variables kept in NIC SRAM, would work correctly. Once the MCP determines that a DMA operation is complete, it updates LR and In as usual. The host could then read LR using PIO, and be assured that the DMA operations were complete as soon as the change in LR was detected. However, since SGI systems allow the DMA transactions to be in-flight for longer than it takes to perform a PIO read, the change in LR could be seen (on SGI systems) before the DMA operation was complete.

So we instead tack a flag on to the end of each buffer and expand it into a four-buffer algorithm. Figures 6 and 7 show the processing performed by the host process and the MCP on the reader side.

The $read$ function uses the additional buffer to provide a pointer to consistent copy of the signal while still waiting for the new data to resolve in memory. Only when the new data has committed to memory is it safe to release the last consistent copy.

The $DMASendToHost$ function in Figure 7 summarizes the processing done by the MCP to transfer data into memory. The $Flag$ value for the target buffer is incremented (so that it does not match the value currently stored in $HostBuff[In].flag$. This value is appended to the data, which is transfered to host memory via DMA. LR is made equal to In, and the In index is incremented and compared against both $Out1$ and$Out2$.

Even though this design does not function correctly on SGI systems (because the last cache line of the DMA operation, which contains the completion flag, may complete before other cache lines), there are three advantages. If the signal buffer plus the status flag will fit in one cache line, the consistency of a cache line will guarantee that the whole DMA transaction is complete when the change in the flag is detected, and therefore the system will work on SGI systems for small signals. Further, this design can be easily modified to include a host interrupt which will guarantee DMA completion. Finally, our tests of SGI systems have shown that while it is very common to read a register via PIO and obtain a result while DMA operations are still pending, it is very rare to observe individual cache lines of a DMA transaction committing to memory out of order. These advantages will allow reasonable performance testing of this approach with and without interrupts.

4.3 DMA with Interrupts

In order to make the previous design work correctly on SGI systems, host interrupts must be used. The reader thread in the previous design examines a flag at the end of the signal to determine DMA completion. Rather than extending the length of the data DMA to transfer the signal data and the flag at once, the MCP can DMA the signal data only. Once the host DMA engine indicates completion, the MCP can trigger a host interrupt. The interrupt routine can then simply examine the current $Out1$ value (via PIO) and set the

```
struct Buffer {
   Signal data;
   int    flat;
}

HostBuff[4];
In = 1;
LR = 0;
Out1 = 0;
Out2 = 0;

Flag[4]; /* initally Zero */

Signal *read() {
  static int awaiting_completion = 0;
  static int correctflag;
  if (!awaiting_completion) {
     if (Out1 != LR) {
       Out1 = LR;
        correctflag = Flags[Out1];
       awaiting_completion = 1;
     }
  }

  if (awaiting_completion) {
    if (HostBuff[Out1].flag == correctflag) {
       awaiting_completion = 0;
       Out2 = Out1;
    }
  }

  return &HostBuff[Out2].data;
}
```

Figure 6: Pseudo-code for Host Receiver Side of Myrinet SNE using DMA.

```
DMASendToHost(Signal data) {

  Flag[In]++;

  /* append Flag[In] to the
     end of the signal data */

  /* Initiate DMA operation of
     signal data plus flag
     to host HostBuff[In] */

  LR = In;
  if (++In > 3) In = 0;
  if (In == Out1 || In == Out2)  {
    if (++In > 3) In = 0;
    if (In == Out1 || In == Out2) {
      if (++In > 3) In = 0;
    }
  }
}
```

Figure 7: Pseudo-code for LANai Receiver Side of Myrinet SNE using DMA.

flag in host buffer $Out1$ equal to $Flag[Out1]$. This accomplishes setting the flag just as the extended DMA operation did, but it is guaranteed not to occur until all cache lines of the DMA transaction have committed to memory.

4.4 Performance

No performance numbers have yet been obtained for these three alternative implementations.

5 Multi-Element Signal Notification Engines

Although signals are typically small, it is important to examine cases in which one signal consists of a large amount of data. The approaches so far described are inefficient if the signal is large but only a small portion of the signal is changed. If a newly produced signal differs from the previous signal only in one byte, for example, the approach given in Figure 1 forces the user write the entire signal structure and transmit it through the network even though only one byte is new data.

Consider a distributed application responsible for tracking the positions and orientations of a large number of physical objects. The data for each object could possibly be handled as a separate signal. But if it becomes important to obtain a snapshot of the system of objects at a given instant, the consistency of the entire set of signals becomes an issue. Under this assumption, combining all the position and orientation data into one signal would be a better approach, if we could avoid sending the entire signal each time.

This suggests that the method used for communicating signals must handle large signals composed of smaller elements, and that these smaller elements must be independently modifiable while still maintaining the consistency of the entire signal. Further, the latency of communication should be proportional to the number of elements which have new data, not the total size of the signal.

Figure 8 provides pseudocode which expands the triple-buffering algorithm to handle signals which consist of a fixed number of uniformly sized elements. Thus the signal becomes a signal vector. Here the writer can make partial updates of the signal vector with the $element\_write(...)$ call, and state that all partial updates are complete with the $end\_write()$ call. The reader captures a consistent signal vector with the $begin\_read()$ call, and obtains pointers to the elements of this signal vector with the $element\_read(...)$ call. These pointers will be valid until the next $begin\_read()$ call.

In essence the the algorithm of Figure 8 performs triple-buffering on each element of the signal vector. The In value is now an array of indices describing which buffer should receive each new element value. The LR value is an array describing for each element which buffers contain data for a new (but not necessarily complete) write. The Out value is an array describing which buffers constitute the most recently received complete signal vector. This allows the reader to assemble the complete signal vector without copying any parts of it. Rather, the indices are copied.

This algorithm can be further extended to provide two additional features important for efficiently extracting information from the signal vector. The first is a mechanism for checking (in constant time) whether or not a given element of the signal vector has changed since the last read. The second is a method for retrieving a list of the elements which have been modified since the last read. Without this feature, the communications latency would still depend strongly on the size of the whole signal, since a user would need to explicitly check each element, even if only only one has changed. We refer to this form of communication as a multi-element signal notification engine (MESNE).

Note that this algorithm has lost some of the desirable features. Since the In and LR are now arrays and must be copied, we must make these sections atomic. When these values were integers, the copy operations were automatically atomic. So some form of blocking is necessary with this algorithm.

```
typedef Element Signal[3][N];

int In[N];  /* all ones initially */
int Out[N]; /* all zeros initially */
int LR[N];  /* all zeros initially */
Set Modified;  /* a set of element numbers */

element_write(int n, Element e) {

  Buffer[In[n]][n] = e;
  Add n to Modified;
}

end_write() {
 int i,in;

 atomic {
    LR = In; /* array copy */
 }

 for (all i in Modified) {
   in = In[i];
   in++; if (in >= 3) in = 0;
   if (in == Out[i]) {
      in++;
      if (in >= 3) in = 0;
   }
   In[i] = in;
 }
 clear Modified;
}

begin_read() {
  atomic {
    Out = LR; /* array copy */
  }
}

Element *element_read(n) {
  return &Signal[Out[n]][n];
}
```

Figure 8: Pseudo-code for sharing a signal vector

Finding an approach that allows partial updates and provides for efficient iteration over new elements without blocking is non-trivial, and we opt instead to find algorithms that may block for small periods, and more importantly, periods that do not depend on the behavior of the network or the partner. However, note that the atomic array copy invalidates another desireable feature: that the communications latency be independent of the size of the signal vector. Since the length of the index arrays equals the number of elements, N, in the signal vector, this copy introduces an O(N) term into the latency. Rather than trying to eliminate this term, we again opt to accept it but keep it small. Since only 2 bits are needed for each index, these arrays can be tightly packed in memory, and the copy operations may consist of only a few word copies.

5.1 Motivation: Computational Architectures to Support Haptic Interfaces

This section provides a real example of a distributed real-time application that would benefit from the use of the described communications moded.

Numerous human-computer interface devices exist for sensing aspects of a user's hand and arm position. These range from simple pointing devices such as a mouse to complex, high degree-of-freedom systems used to recognize hand gestures in virtual environment systems. Most such interfaces are pure input devices in that they do not provide the user with a sense of either touch or force. In contrast, *haptic interfaces* not only measure position of the user's arm or hand, they are capable of generating forces which can be felt by the user [ND94].

The first use of haptic interfaces was for teleoperations applications. In teleoperations, a user manipulates a remote physical device through some sort of interface. Often, the remote device is designed to grasp and manipulate objects. Providing a user with the sensation that he or she is actually touching and moving objects at the remote location has been shown to significantly aid in teleoperations tasks [TS92]. Haptic interfaces achieve this effect by providing a two-way interaction between the user and the physical environment. Hand and arm positions of the user are sensed and used to control actuators in the teleoperated device. As the teleoperated device contacts objects, sensors measure the forces that result. Actuators in the haptic interface are used to *reflect* these forces to the user.

Figure 9 illustrates the physical architecture used to implement force reflecting teleoperations systems. Both the haptic interface and the teleoperated device need sensors, actuators, and controllers. Modern systems use digital controllers to perform the functions of sampling sensors, sending the values to the companion device, receiving values from the companion device, and controlling actuators. Humans have surprising tactile acuity [MM90], so in order to provide a realistic sense of contact and force, these must implement servo loops which repeatedly sense position, determine object contact, determine resulting haptic forces, and exert those forces, and these loops must be run at kilohertz rates. Except when the interface and the controlled device are separated by significant distances, communicating between the two is typically done via some sort of dedicated channel. This minimizes communication latencies which would otherwise degrade the intended perceptual effects.

Recently, haptic interfaces have been used to allow interaction with simulated physical entities in a virtual environment. When coupled with visual displays, they can provide a user with a much more compelling sense of immersion in a virtual world than would be possible using visual displays alone. This is

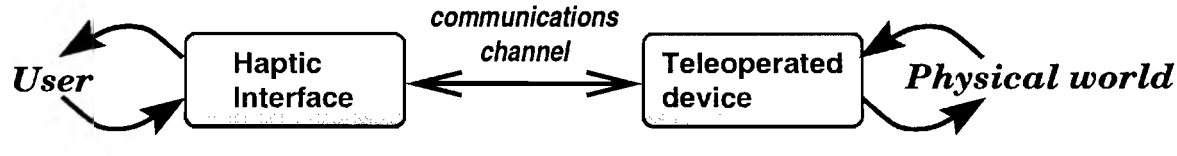
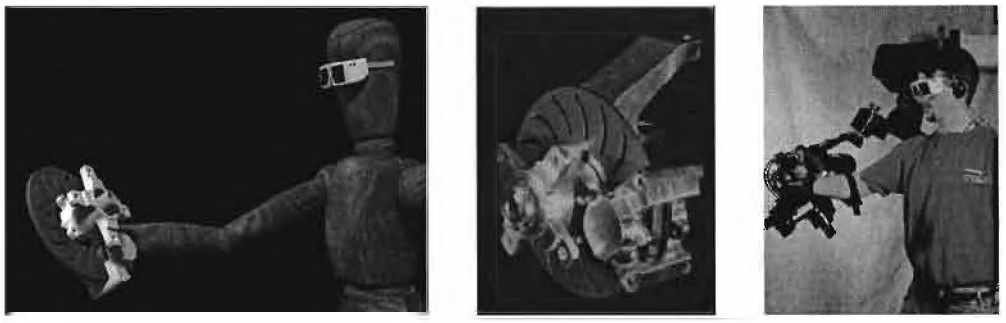


Figure 9: Haptic interfaces for teleoperations.



(a) (b) (c)

Figure 10: Depiction of a user employing a haptic interface (c) to interact with a virtual brake assembly (a). A physical instance of the actual brake assembly is shown in (b).

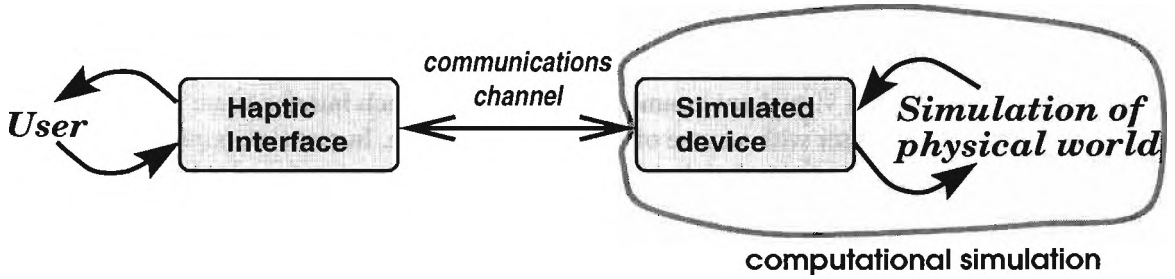


Figure 11: Haptic interfaces for virtual environment.

particularly useful in design applications where it is desirable to experiment with user interactions with a contemplated assembly without the need to construct a physical prototype (see Figure 10) [JMH97]. Haptic interfaces to virtual environments utilize the same sorts of sensors, actuators, and controllers as do force-reflective teleoperations systems. Instead of controlling a real teleoperated device, however, they interact with a computational simulation. The simulation takes in information about the user's hand and arm positions, simulates the effect that would occur if a real teleoperated device was being controlled, next simulates the changes that would occur in the environment based on the predicted movement of the teleoperated device, and finally simulates the forces that would be generated on that device for reflection back to the haptic interface actuators (Figure 11).

Significant challenges must be overcome in implementing the architecture shown in Figure 11 in a way that achieves adequate performance. The 1+kHz servoing rates needed for the haptic interface mandate digital controllers operating under a real-time OS and typically running on some sort of micro-computer. Simulations of the physical world often require prodigious amounts of computational power. This, plus the frequent need in these simulations for interaction with legacy software, means that the simulations need to be run on high-capacity compute servers. Such servers seldom support either deterministic real-time scheduling or dedicated communication channels, both of which would be required to achieve the necessary cycling rates using an architecture such as shown in Figure 11. Even if the OS and communications latencies could be sufficiently reduced, many computational simulations of physical effects cannot be processed completely in the time needed to support perceptual realism in a haptic interface.

Figure 12 illustrates an alternate computational architecture with the potential to overcome at least some of these difficulties. The key is to move a portion of the simulation calculations onto the real-time system controlling the haptic interface. This allows the simulation code to be run under a real-time OS and gains the added benefit of avoiding the communications overhead inherent in exchanging information between

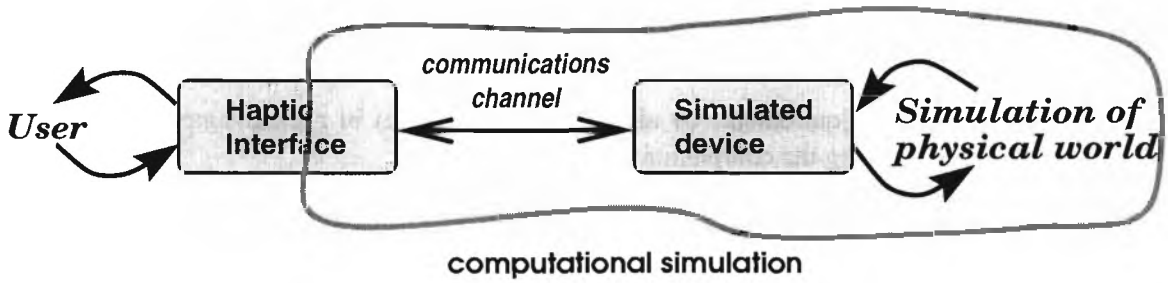


Figure 12: Alternate architecture implementing haptic interfaces for virtual environment.

heterogeneous computer systems. Since the computing power of the real-time system will be limited, it is essential that *only* those portions of the simulation critical to achieving perceptual realism be moved. This requires that the necessary information be sent from the compute servers to the real-time controller for caching and that the cached information be updated in a timely manner so as to preserve the realism of the simulation.

Running a portion of the simulation on the real-time controller significantly complicates communication and synchronization within the overall system. State information has to be exchanged between the real-time system and the compute servers to insure consistency in a simulation that is now distributed across disparate computational engines. Synchronization problems are made more difficult by the fact that the real-time system will typically be running control loops with cycle times much faster than can be achieved with the more complex computations being run on the compute servers. Those portions of the simulation running on the computer servers may themselves be distributed computations resulting in a behavior in which different aspects of the state of the simulation are updated at different times.

We can abstract the requirement for communications between the real-time controller and the computer servers into the following set of constraints:

- Communication involves exchange of state vectors between two processes.
- Readers and Writers need to be able to operate asynchronously.
- Writers generate different aspects of state information at different rates.
- Readers care only about the most recently produced state.

The simplest way to satisfy these requirements would be for entities which need state information to explicitly request it from those entities where the needed information is computed. There are two significant potential sources of latency in this communication model. (1) A round trip message needs to be sent through the communications channel. (2) Generating the requested information can take a significant amount of time. The second effect can be reduced by continually generating relevant information and then caching the results of the most recently completed computation. While worst case response time will be the same, average case response will be halved. The round trip communications latency can be halved by doing the actual caching of state at the receiver. If different aspects of state become available at different instances in time, the communications bandwidth needed to transfer state updates to the receiver can be reduced if the communications protocol supports the transmission of partial information about state. Since the consistency of the state vector is important, we also require a mechanism to group these partial updates into one atomic modification of the state.

Summarizing these requirements:

- Minimize communication latencies by caching most recently generated state information at the receiver.
- Minimize bandwidth requirements by allowing partial updates of cached state information, with a mechanism for signalling the completion of all partial updates.

5.2 The MESNE Communications Model in Detail

Section 5 examined a simple algorithm in shared memory providing a first attempt at the desired communications model. This section expands that model, describing the form of the user interface and its functionality, but avoids discussion of the algorithms necessary to implement an ideal MESNE communication system.

MESNE allows a writer process to communicate a signal vector to a reader process in such a way that the writer can make any number of partial updates to the elements of the vector and then mark the signal consistent. The reader will then receive only the most recent consistent signal vector.

A MESNE signal vector consists of N elements of S bytes each. The signal vector is therefore an opaque memory region of $N * S$ bytes which is segmented into N uniform elements. For two processes to communicate using MESNE, they must agree on the configuration (N,S) of this signal vector, and one must act as writer and the other as reader.

The writer API has three fundamental calls: $mesne\_begin\_write(...)$, $mesne\_element\_write(...)$, and $mesne\_end\_write(...)$. The $mesne\_element\_write(...)$ call assigns new data to a specified element of the signal vector. This can be called any number of times for any elements of the vector, and each call is interpreted as a partial update of the signal vector. By calling $mesne\_end\_write(...)$, the writer is stating that these writes are now complete. The signal vector is then in a consistent state, and can now be published to the reader. After the $mesne\_end\_write(...)$ call, the system is assumed to be in a state unsafe for writing as it works to ensure that the reader has the correct values for each element. The $mesne\_begin\_write(...)$ call will block until the system is ready for writing again. The begin and end calls thus act as parentheses for the writes. All writes inside a begin/end pair will be treated as one write. These calls also serve to hide any blocking that may occur because of work that must be done by other processes in the system. (Although $mesne\_begin\_write(...)$ can block, we insist that it not block for any reason involving buffer capacity or actions of the reader process. Only work remaining for the local host can delay the return of the $mesne\_begin\_write(...)$ call.)

The reader API also has a begin/end design. The $mesne\_begin\_read()$ call provides the reader with a snapshot of the most recent complete signal vector, i.e. one with all previous completed writes applied. This snapshot can be examined with the $mesne\_element\_read(...)$ call, which requires an element number and returns the current data contained in that element. It also provides the reader process with an indication of whether or not this element's data is new as of this begin/end pair. This snapshot is guaranteed to be consistent and static, regardless of any further updates from the writer, until the read is completed using the $mesne\_end\_read()$ call. The next $mesne\_begin\_read()$ will block until the signal vector is consistent and ready for examination. (Again, the $mesne\_begin\_read()$ will only block to await the completion of work done by the local host. No actions taken by the writer will delay the return of $mesne\_begin\_read()$.

Since the signal vector may be large, and only a few of the elements may contain new data, we provide a method for iterating over only those elements which are new. The $mesne\_iterate\_read(...)$ function accepts an iterator function and will call this function for each element that has new data, supplying it with the element number and a pointer to the new data. The $mesne\_iterate\_read(...)$ call will return the number of elements in the signal vector which contained new data. This mechanism prevents the user from having to examine each element in the signal vector.

Note that there are no synchronization requirements between the partners. The two processes can perform their reads or writes at any rate and at any time, without risk of buffer overflow or blocking for long

periods of time.

The configuration (N,S) is important because it specifies the granularity of both the writer's partial updates and the reader's indication of which portions have changed. One element of S bytes is the smallest portion that can be partially updated by the writer, and the smallest portion over which the reader can detect a change.

Briefly, this functionality is achieved as follows. A writer process keeps a pool of buffers, each equal in size to one element (S bytes) to accept partial writes. For each element that is updated, the writer copies the data into one of these buffers, and updates indexing data structures. All of these updates are considered pending until the writer process calls $mesne\_end\_write(...)$. This marks the signal vector as consistent.

A network process reads from these buffers and sends the data to the partner. The buffering allows the writer to continue writing even if the network is blocked and cannot send. So, many complete updates to the signal vector can be collapsed in these buffers. When the network unblocks, it can begin sending only those packets which are a part of the most recent consistent signal vector. When all new elements have been sent, the network process sends an end-of-write packet (EOW).

The reader keeps a pool of buffers, each equal in size to one element (S bytes). It is the responsibility of the receiving code to distribute the incoming data packets among these buffers and properly update internal indexing data structures upon receiving an EOW so that it can always reconstruct the most recently received consistent signal vector, and can always receive more packets. The reconstruction should not introduce unnecessary copies of the data, and since old data is not significant, the required buffer space is finite.

Note that this system allows each element update to be sent to the reader immediately, without waiting for the signal vector to become consistent. By eagerly sending data to the reader and making the reader responsible for applying the updates and enforcing consistency, we reduce the latency of communication.

Two processes establish a connection with each other using $mesne\_connect\_read(...)$ and $mesne\_connect\_write(...)$, each specifying the other as the target of communication. Either partner can perform this connect first. Even if the partner is not present, the rest of the API calls function identically. Since only the most recent data is important, MESNE treats an absent writer process in the same way as one that is not sending data. Writer processes can send data regardless of whether or not a reader is present, with the assurance that if one does appear, it will receive the most recent complete signal vector. Each partner can therefore drop out of the connection at any time without affecting the other, and can reestablish communication by using the connect call again.

5.3 Prototype Myrinet Implementation

The Myrinet SNE designs were simple, and it was easy to argue that the latencies for such designs should not be lower than that of a well-implemented packet-stream protocol. However, MESNE is considerably more complex, and it is not clear that implementing the majority of the functionality using Myrinet's processor is the correct choice for the lowest latency solution. [2]

The remainder of this paper describes a prototype MESNE implementation intended to explore the design space. Rather than attempt to implement a fully functioning MESNE natively on Myrinet hardware, we first implement only the reader half, using a simple send queue for the writer, to determine if the latency is low enough and the design task simple enough to continue with a full implementation. This simple prototype also allows us to explore two design trade-offs concerning reliability and receive-buffer placement.

[2]It is true that using Myrinet's processor removes the burden of handling packet reception from the host processor, but the host processor is considerably faster and the savings may be inconsequental. Possibly more important is the fact that a Myrinet MESNE solution on SGI systems would require only one interrupt per complete write (which may consist of many individual element updates), rather than one interrupt per packet received. This is because the reader is guaranteed not to examine the data until the write is complete. With more generic packet transmission protocols, the user must be able to examine each received packet.

The algorithm presented in Figure 8 should serve as an example of the general methods we will use to provide the functionality of the ideal MESNE communications model. We will not discuss the necessary algorithms in detail here; they are detailed fully in Appendix B.

Although the prototype has simplified the writer's end of the communication, all other parts of the prototype are fully implemented, providing protected multi-user access, the complete reader's end API, and network routing tools. The full documentation of the prototype MESNE API is given in Section Appendix A.

5.3.1 Design

The prototype implementation of MESNE consists of custom LANai software, a device driver, a user level library, and routing and initialization scripts. In this section we discuss the high level design of MESNE. More detailed design and implementation information is provided in the appendices as noted.

5.3.2 The Communications Endpoint

The MESNE prototype provides multiple user processes with protected access to the network hardware by partitioning the NIC SRAM into a number of communcations endpoints, called ports. A user process establishing a MESNE connection `mmaps` one of these endpoints into its address space, allowing direct programmed I/O (PIO) access to the network hardware for control. The LANai software handles multiple endpoints in round-robin fashion.

For a writer, the communications endpoint in SRAM contains status and control variables, a shared queue, and an area for writing element updates. This provides the writer with a queue-based interface to the network. A writer process inserts its element updates into this queue, and the LANai software extracts and delivers them to the communications partner. (Note again that this is a simplification from the ideal MESNE design, and blocking can result if the network cannot empty the queue faster than the writer process fills it.)

For a reader, the communications endpoint consists of an SRAM protion and a separate DMA capable portion of host memory. The SRAM contains status and control variables, And the host memory region is used by the network to store individual element updates.

It is important to note that the connect functions in the MESNE API establish connections between two ports on two separate network interfaces, not between two processes. In order to communicate, two processes must each claim a port on their host and specify a connection to the remote partner's host and port. These two ports are then claimed for the specified communication. When the other partner attempts the reciprocal connection, it will simply join the one already established. Once the two communicating partners are joined, either partner may disconnect without disturbing the other; i.e.,a reader whose partner writer has released will simply receive no new data, and a writer whose partner reader has released will simply be unaware that no one is paying attention to the data it sends. The connection between the two ports will not be dissolved until both partners disconnect or terminate.

5.3.3 Sending Packets

In order to communicate a signal vector with configuration (N,S), N elements of S bytes each, the MESNE API provides a writer with a writeable buffer of $N * S$ bytes (in the endpoint SRAM) which holds the current value of the signal vector. Any partial update to the signal (i.e. modification of an element) is made directly to this buffer by the API library. Since this buffer is kept in NIC SRAM, all writes to this buffer are done via PIO. Therefore, all host-to-card transfers are done via PIO, as with Fast Messages [SP97].

The NIC also maintains a queue of elements which have been written. When the user writes an element, the API library modifies the element using PIO, and then enters the number of the element in the queue.

The LANai software reads element numbers out of the queue, and sends a network packet to the reader consisting of the element number and the data currently stored in that element. When the user ends the write, thus stating that the signal vector is consistent, the LANai will send an end-of-write (EOW) packet when the queue is empty.

5.3.4 Reliability

Many communications protocols and networks provide reliable ordered delivery, but the assumptions required for such reliability can differ. For example, TCP/IP provides reliable delivery between two host computers even if hardware along the communications path fails, or the network topology is changed but still provides a route. Fast Messages for Myrinet, on the other hand, makes more restrictive assumptions about the hardware to achieve reliable ordered delivery.

Since Myrinet is inherently very reliable, the chance of a lost or corrupt packet due to a bit error in a data byte, routing byte, or internal control byte is very small and can be comfortably ignored. If cables are disconnected or switches lose power, packets can also be lost or corrupted, but many users of Myrinet safely assume these events will not occur. But because Myrinet was designed to be able to clear packet deadlocks autonomously, there is an additional possibility for lost packets. If a packet is blocked at the input port of an interface card or a switch for longer than a specified time (50ms in current generation Myrinet components), the packet will be dropped.

This delay can occur at an interface card if a packet arrives from the network when no memory is available to store it. The DMA engine which transfers packets from the network to NIC memory will be idle, and the packet will block. When a receive buffer is made available in NIC memory, the DMA engine can be started and the packet will be received.

Fast Messages (FM) achieves reliable ordered delivery as follows. It assumes the network topology is fixed and consistently powered, and that the bit error rate is practically zero. It eliminates the final possibility for dropped packets by employing a send-credit scheme. When a connection is formed, the sender is granted a number of send credits proportional to the amount of buffer space the receiver has available for that connection. Each sent packet deducts a credit. When the credits are gone, the sender is not allowed to transmit any more packets. Only when the receiving host computer consumes the received packets and thus frees buffer space is credit returned to the sender. So, packets will never be sent to an interface unless buffer space is available, and therefore no packet will ever have to wait. We will refer to this form of reliability, which requires assumptions about the network hardware, as FM reliability, and the TCP/IP form we will refer to as full reliability.

With the MESNE prototype we have explored two design options concerning reliability. One is to ensure full reliability by employing a common sliding window algorithm (explained in Appendix C) on transmitted packets. The other is to make no effort at ensuring reliability and examine the design to see if the requirements for FM reliability are satisfied.

5.3.5 Receive Buffer Space Requirements

The LANai software on the receive side maintains a pool of S-byte buffers to receive these data packets. It must direct the data into these buffers and adjust internal data structures so that the most recently received consistent signal vector can be reconstructed and made available to the user for an indefinite period of time. It must also be able to receive any number of further data or EOW packets without running out of buffer space or corrupting the reconstructed signal vector.

The algorithm presented in figure 8 shows that using 3N buffers is sufficient if shared memory is used as the underlying communications medium. However, since a write is a network operation, and because low-latency is an important goal, there are compelling reasons to use 4N buffers. To summarize, the 3N-

buffer design can fail if packets can be corrupted, since it expects only accurate data to be written into each buffer. Also, the processing which must be done in the $end\_write()$ call in Figure 8 can increase the latency of signal communication, and a 4N-buffer algorithm can reduce this work. (These issues are explained in detail in Appendix B.)

Further, additional data structures are needed to direct the reception and to allow the reconstruction of the most recently received consistent signal vector whenever the reader requests it. Although 8 gave a simple example of the kind of processing we need to do, i.e. making In, Out, and LR arrays of indices describing the correct buffer locations for reading or updating, the actual algorithm that we use is more complex and is described in Appendix B.

Since MESNE is a sender-based approach, the NIC must manage these data structures, and so they are kept in LANAi SRAM. However, the pool of 4N buffers need not be kept there.

5.3.6 Receive Buffer Locations

There are two possible choices for the location of this pool of 4N buffers. First, we could keep it directly in the NIC's memory. The NIC software can simply examine the packet header, consult its data structures, and receive the rest of the packet into the correct buffer. This would mean that any packet from the network would always have a buffer on the NIC available to receive it. Therefore no packets would have to wait, the 50ms timeout would never occur, and no packets would be dropped. Thus, simply because the MESNE communications model never requires more than 4N receive buffers of S bytes each, we can ensure FM reliability with no additional effort. [3]

There are two drawbacks to this choice. First, NIC SRAM is fairly scarce. This choice limits the size of the largest signal vector that can be communicated and the total number of connections that the interface can handle. The second drawback is that when the reader requests the most recent consistent signal vector, it should be available directly in the memory of the reader process, not in the NIC SRAM, so that reads of the signal vector do not have to access the PCI bus (as they did in the simple PIO version of the myrnet SNE). The LANai software must respond to this request by determining which buffers contain the correct element data and transferring them one at a time to a reassembled signal vector in the reader process's memory. Although this DMA from the NIC to the host is a necessary copy, more latency is introduced because this transfer must be started only after the reader process requests a new signal vector.

The second choice is to place this pool of 4N buffers in the reader host's memory. The NIC can simply consume all incoming packets from the network into a circular buffer. It can then empty this buffer by examining the packet and its data structures to decide where it belongs in the reader's process memory, and initiaing a to-host DMA to the correct user buffer. When the reader requests a the signal vector, the NIC can simply supply it with a list of indices indicating which buffers in this pool form the most recently received consistent signal vector. Since the network will then simply serve to allow a remote writer to make modifications to buffers in the reader's memory, this scheme is very similar to the triple-buffering algorithm with all writes occuring over the network.

The advantages are that the net receive engine and the to-host engine are more likely to operate simultaneously, and the work required to supply the reader with the signal vector is much simpler. These both reduce the latency. However, we can no longer guarantee that a packet will not wait to be received by the NIC, since the circular buffer can fill. The time required to free enough space in a full circular buffer to receive an arbitrary packet is dependent on the workload of the LANai and the to-host DMA characteristics and is not necessarily bounded. Therefore, this option is not FM reliable.

[3] Note that Myrinet switches can also drop packets. This will occur if a packet arriving at one switch port must wait longer than 50 ms for its desired output port. For large busy networks with bottlenecks, these losses are a real possibility, and FM could not be considered reliable in these cases.

Figure 1 summarizes the resulting reliability of each of the four design options. Each of these options have been implemented in the prototype, and we compare the test results in section 5.5.

	No Sliding Window	Sliding Window
Buffers in NIC SRAM	FM reliable	Fully reliable
Buffers in Host Memory	NOT reliable	Fully reliable

Table 1: Resulting reliability of the four design options.

5.3.7 The Device Driver

To avoid the latencies incurred with system calls, the device driver does not participate directly in sending or receiving data. The device driver handles the allocation of the multiple communications endpoints, called ports, on each interface, so that multiple user processes can access the network. It controls user-level access to the limited number of ports on each interface, allocates portions of the LANai SRAM and a DMA capable portion of host memory for use by each user process using MESNE, and regulates `mmap` access to these locations. It is also responsible for cleaning up these resources when user processes disconnect or crash.

When a user process calls *mesne_connect_write* or *mesne_connect_read*, the device driver claims the specified local port and attempts to forge a connection with the remote port on the remote interface. If the local port is available, and the remote port is either unused or in use by a suitable parter, the device driver allocates a portion of LANai SRAM and a DMA capable region of host memory for the exclusive use of this process. The device driver finally allows the user process to *mmap* these areas, and all communication with the NIC occurs through these mappings. When the process calls *mesne_disconnect*, this function unmaps all NIC and host memory, and this informs the driver to tear down the connection and release all NIC resources. Appendix D explains in detail.

5.3.8 Routing Issues

We assume that the network topology is static. Each interface is assigned a unique interface number, and routing instructions for each interface are generated by hand and kept in a human-readable file. On boot up, the host must initialize the NIC with a routing table and assign it a unique interface number. An initialization script examines the routing file and initializes all interfaces on the host.

5.3.9 The custom LANai software

In this implementation, the LANai software performs nearly all the processing required by the protocol (described in section Appendix B). It is divided into an interrupt context routine and a user context routine. The interrupt context routine handles all packet receptions. If the buffer pool is kept in the host, this routine also handles the DMA which transfers the packets to the correct host buffers. The user context routine handles sending packets for all writer ports and the processing of all end-of-read signals from reader ports.

Processing for the writer ports simply involves consuming element numbers from the shared queue and sending the data, already present on the LANai, through the network. It responds to a *mesne_end_write* command from the writer process by emptying the queue and sending an end-of-write packet to the reader. Processing for reader ports occurs when the *mesne_end_read* function is called, ending the previous read.

The LANai updates the data structures and host data buffers so as to provide the reader process with the most recent consistent signal vector.

The LANai software also establishes and tears down connections with remote ports when requested to do so by the device driver, but the methods used are standard and will not be described here.

5.3.10 Determining DMA Completion

As discussed in section 3.2, since SGI systems allow multiple cache lines of a DMA operation to commit to memory out of order, an interrupt is required to be certain that a DMA operation has completed. However, the re-ordering of cache lines in a DMA transaction is highly unlikely. The MESNE prototype can signal completion of a DMA operation by detecting a change in the status word located one word past the end of the DMA's target location. This status word can be changed either by making the DMA operation one word longer or by using the host interrupt routine.

5.4 Related Work

Virtual environment researchers have addressed their needs for low latency communication in a number of ways. The designers of the CAVE Virtual Environment [CC92] chose SCRAMnet [TB98], a broadcast memory system, for their communications medium. SCRAMnet provides simple replication of local memory writes to the memory of other hosts on the network. Implementinga communications model such as ours over SCRAMnet would not necessarily be trivial, since efficiently implementing synchronization and mutual exclusion over such a replicated memory system is an area of research [SM98].

Other shared memory systems are available which provide true distributed shared memory to heterogeneous computing elements. Although the latencies of these systems are impressive, they require a significant investment in special purpose hardware. However, as discussed, MESNE can be seen as a set of shared memory algorithms adapted for use over a LAN; therefore, MESNE can be easily designed to use shared memory between two processes whenever possible.

The Network Data Delivery Service, NDDS [GP94], provides another example of a most-recent-only communications paradigm. With NDDS, hosts on a network "publish" the signals they produce and "subscribe" to signals they need. In addition, under NDDS, signals may be published and received by any number of hosts. Each publisher is rated with a strength and duration, and receivers will receive only the strongest signal whose duration has not expired. This fits well with most fault-tolerant real-time systems, since the signals produced by "hot standby" publishers will automatically be used if a primary publisher fails. Since it uses multicast UDP/IP as its underlying communcations medium, it easily supports many-to-many communcation and is inherently portable; however, the typical latencies are double that of UDP for small packet sizes.

Like the prototype MESNE implementation, the rest of the approaches which we will discuss have also used Myrinet as the interconnect fabric, although not necessarily exclusively. These approaches have either inspired certain aspects of the prototype's construction or could be used as underlying communications protocols constructing a communications model similar to MESNE. We will use these to compare implementation details and performance results. For an excellent overview of current protocol research using Myrinet, see [RAFB98].

The basic structure of the Myrinet MESNE prototype is similar to Direct Deposit [MRS95], Hamlyn [GB95], VMMC [CD97], and other sender-based protocols which use a shared-memory abstraction to eliminate receive buffer overflow problems and reduce OS invovement. In each of these, a sender is allowed to write directly into a receiver's buffer by specifying the target address within this buffer. Direct Deposit is connection oriented, follows a client-server model, and supports bidirectional communication. In this protocol, each write into the receiver's memory generates a notification which is appended to a queue. These

notifications inform the receiver of new data at specific locations in its receive buffer. However, programs that receive data at a faster rate than they consume from the queue run the risk of losing notifications. Therefore the user must provide some synchronization. Hamlyn is similar to Direct Deposit but has provisions for preventing notification queue overrun. It also provides key-based network security and support for adaptive routing networks. VMMC makes notifications optional. They can be attached to a message, causing the invocation of a user-level handler function after the message has arrived in process memory. Thus there is no explicit receive operation.

The user-level orientation and multi-user support provided in the Myrinet MESNE prototype follows that of U-Net [TE95] and others [RAFB98], which provides user-level access to various network interfaces, including Myrinet, ATM, and fast ethernet. Such user-level access removes the operating system from involvement in packet sending or receiving, thus eliminating system call overhead. Although the basic U-Net implementation simply provides a user-level entry point into the network for sending and receiving a stream of packets, a superset called Direct-access U-Net allows a sender to place a packet directly in the receiving process's address space at an offset specified by the sender, and provides another example of a sender-based protocol.

The Myrinet MESNE prototype is similar to all of these in that an incoming packet is placed directly in receive buffers by the NIC based on the directions of the sender, without waiting for an explicit receive operation. However, the prototype's receiving software interprets the incoming packets as modifications to a signal vector and distributes the receivied packets into a pool of buffers in such a way that it can always reconstruct the most recently received consistent signal vector and can always receive more data packets. So the MESNE prototype is similar to a sender-based protocol with a more complex receive mechanism.

The MESNE prototype provides the required high-level functionality in one protocol level, whereas the others provide low-level interfaces of much more general use. It would be difficult, for example, to implement another protocol on top of this prototype.

Fast Messages (FM) [SP97] is a messaging layer specifically designed to support the implementation of higher level messaging layers. It is intended more for language and library designers rather than end users. It provides a reliable ordered message stream, decoupling of communication and computation (provided by message handlers which process packets only when the user requests), and freedom from communication deadlock. The Myrinet implementation of FM uses a flow control technique to obtain reliable delivery over Myrinet. We will discuss this technique and compare it with a simpler approach that one version of MESNE allows.

Put/Get on FM [JN96] and Global Arrays on FM [LAG98] are examples of more sophisticated shared-memory protocols which have been implemented over FM. Both provide a rich set of primitives for distributed computing. Put/Get enables a group of distributed processes to read and write the virtual memory of any process in the group. It provides process synchronization through barriers and atomic read-and-set operations on single integer values. Global Arrays allows global floating-point or integer matrices to be shared among distributed processes, and allows the process to determine which portions of a distributed matrix are stored locally. Each process can read or write a submatrix of the global matrix asynchronously, without requiring the cooperation of any other process. Synchronization is provided by process barriers, distributed mutexes, and by atomic read-and-increment operations on matrix elements.

Put/Get and Global Arrays are high-level protocols implemented over FM's more general low-level interface. This will present a valuable comparison to MESNE's approach of implementing a special purpose high-level protocol in one layer.

5.5 Performance

Latency tests were performed on a four-processor SGI Onyx2 holding two Myrinet interface cards. A single process opened a connection as writer on one card and a connection as reader on the other. The processor

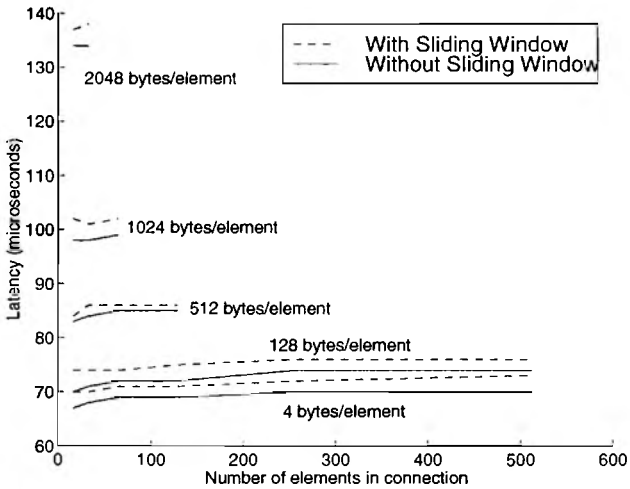


Figure 13: MESNE Version 1. Latency for a write with only one new element.

was restricted from running any other processes and isolated from handling OS functions (such as TLB miss servicing, device interrupt handling, etc.), and its clock scheduler was disabled, thus allowing the test process exclusive and uninterrupted use of this processor. The test process repeatedly wrote its elements and called $mesne\_end\_write$ on one interface, and then called $mesne\_begin\_read$ and $mesne\_end\_read$ on the receiving interface until the new signal vector was available. The elapsed time was measured with the free-running hardware counter, which has a resolution of 0.8μs. This approach allowed accurate timing of one-way latency, and accessing only one card at a time prevented PCI bus contention. This test method is comparable to the test used to determine the minimum latency of 105 μs for the MyriAPI. A round trip test was used to determine the 50 μs one-way latency of the GM API on these platforms. The MESNE prototype was set to signal completion of a DMA operation by placing a status word at the end of a DMA operation. Thus, interrupts were avoided. Although DMA reordering events are extremely rare and have not been detected, this signalling option does not guarantee correct functioning. If interrupts were selected instead, the implementation would be correct, but the latency results given here would be considerably higher.

We emphasize that these test conditions eliminate many extraneous sources of latency and variation: context switching, all OS operations, and use of the NIC by other processes. We report the minimum times revealed by these tests and note that the average times obtained were rougly 20% higher.

5.5.1 Version 1: Buffer Pool in NIC SRAM

The first implementation of MESNE placed the receive buffer pool in the NIC SRAM. In this version, the NIC updates a signal vector in the reader process' memory from this buffer pool using DMA whenever a new signal vector is requested. This version was tested both with and without a sliding window algorithm which guaranteed reliable transmission. Recall that, without the sliding window algorithm, this version is FM reliable.

Figure 13 shows the latency for receiving one new element per write as a function of the number and size of the elements in the connection. Note that in this graph, the number of elements in a signal vector is more limited for larger element sizes because of the limited memory of the LANai.

The latencies for this version are considerably higher than that of GM's at 50 μ. However, certain desirable properties are evident. The latency for communicating one new element of data per write increases very slowly with the total number of elements in the signal vector, and the sliding window adds only a small overhead of about 4 μs/packet to the latency. Since in this version the data is not transfered into host

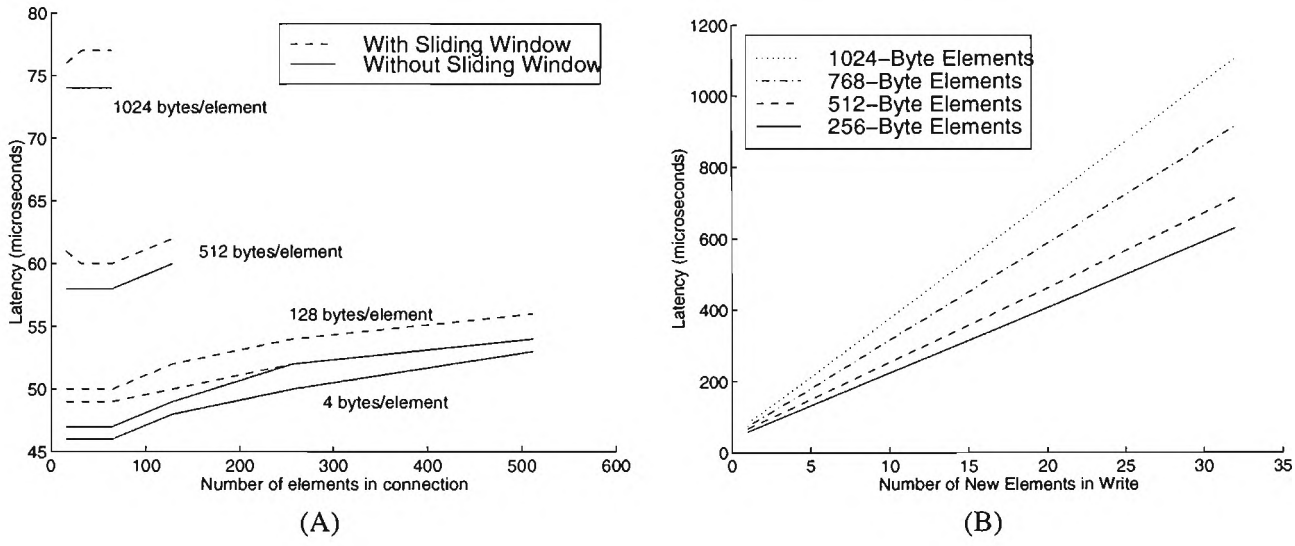


Figure 14: MESNE Version 2. (A) Latency for a write with only one new element. (B) Latency as a function of the size of the elements and the number of new elements

memory until after a new signal vector is requested by the reader, and therefore the net receive and to-host DMA engines cannot act simultaneously, we expect these results to be higher than those of version 2.

5.5.2 Version 2: Buffer Pool in Host memory

In this version, the buffer pool is kept in host memory. Data is placed in the reader process' memory as soon as it is received from the network.

This version achieves a minimum latency comparable to GM. Figure 14 A shows that reliability is still achieved with a cost of rougly 4μs per packet. In addition, Figure 14 B shows that the latency of updating several elements per write is linear in the number of updated elements. The additional cost per element varies from 18 μs/element with 256-byte elements to 33 μs/element with 1K elements.

5.5.3 Discussion

Although placing the temporary receive buffers on the NIC provides some advantages in terms of reliability, it is clear that the latency cost of this choice is much higher than implementing a standard sliding window scheme and placing receive buffers in host memory. However, even this choice is not enough to reduce the latency results below those of GM. In order to make this implementation correct and complete, interrupts must be added and the writer's side must be fully implemented. These changes can only increase the latency, and so we conclude that MESNE is better implemented using GM or another available protocol as the underlying communications layer.

5.5.4 Performance Comparison to Related Work

Current work in low-latency protocols over Myrinet is impressive. Many protocols have minimum latency measures between 5 and 50 microseconds. Basic Interface for Parallelism (BIP) [PTW98] provides a very simple but high-performance API which has achieved a remarkable 5 μs latency for 4 byte packets. Direct Deposit has achieved a one-way latency of 30 μs for small packets on HP J210 machines. Hamlyn, on HP J200 machines, has achieved a minimum latency of 13 to 30 μs, depending on the mode of operation.

VMMC and FM have achieved 10 μs and 11 μs minimum latencies, respectively, on PCI PC's. The minimum latencies of put and get operations for Global arrays over Myrinet are 7.4 μs and 33 μs respectively on PCI PC platforms.

Although none of these protocols are directly comparable to MESNE in terms of memory-protected access and provisions for data consistency, either providing none of these features or much more general forms than MESNE provides, we must attempt to compare the latencies to judge the quality of our prototype MESNE implementation. Hamlyn, Direct Deposit, and VMMC do not provide synchronization or atomic access to the receive area, so we expect their latencies to be lower than MESNE. While the DMA performance of PCI PC's is in general superior to that of the SGI architectures, the HP architecture is comparable [GB95]. So, nearly all the difference between the performance of Direct Deposit or Hamlyn and MESNE is due to the protocol and implementation. BIP does not provide multiple user processes protected access to the NIC.

Put/Get and Global Arrays provide synchronization, but unlike MESNE, neither directly provides the reader with unlimited atomic access to the entire shared region without blocking further updates by the writer. Such functionality could be implemented easily using these systems by arranging mutually exclusive access to a set of data structures and triple-buffering the data, but this would require several atomic operations and shared variable accesses in addition to the data communication. MESNE avoids these by handling all consistency issues on the reader side. Lacking latency data for the atomic read-and-update operations of these protocols on hardware that is comparable to MESNE's, we cannot compare directly. However, MESNE on comparable hardware is likely to have lower latency simply because it would consist of one protocol layer rather than three.

These performance measurements show that for the smallest packets, MESNE's latencies are comparable to GM's. Futher, MESNE, as implemented directly in the LANai, can process all arriving packets so that absolutely no host processor activity is needed to handle individual element updates reception. This is especially important when the writer is sending much faster than the reader is reading. FM must call a user-level handler function to process incoming packets. Similarly, a GM implementation would require host processor activity to accept packets and provide the interface with new receive buffers. MESNE avoids this small but unnecessary drain on the host processor. Although this property might not show up in latency measurements, even a small drain on potentially heavily loaded real-time computation engines may be worth avoiding.

This implementation achieves low latency at the cost of bandwith, since the prototype uses PIO instead of DMA for host-to-NIC data transactions. Figure 14 shows that 32K can be transmitted in rougly 1.1 ms, giving a bandwidth of 28.4 MB/s, which is about 75% of the limiting PIO bandwith. Portability has also been sacrificed; an implementation built on top of GM would be portable to any platform supported by GM with a simple recompile, whereas the current MESNE prototype would be difficult to port to another platform.

6 Future work

Since the Latency of our prototype MESNE design is close to the latency of GM, and the addition of interrupts is likely to increase the latency further, it will be more productive to implement MESNE using GM as the underlying layer, when GM drivers become available for SGI systems. TCP, UDP, and shared-memory implementations would also make the design more useful.

Although the simple SNE designs lack some of the desireable features of MESNE, we can easily construct a complete native Myrinet SNE implementation by simplyfying the MESNE prototype. For packets smaller than a cache line, where interrupts are not needed, this approach will certainly have lower latency than GM, and may be useful for various projects.

7 Summary

The SNE and MESNE designs provide a simple method which allows a writer process and a reader process to communicate the most recent value of a signal. The SNE design allows communication of one monolithic signal. The MESNE design allows the writer to make any number of partial updates to this signal vector or mark the write complete at any time. The reader is allowed unlimited atomic access to a copy of the signal vector with all previous completed writes applied. The asynchronous nature of these protocols frees the programmer from worrying about the relative rates of sending and receiving and also allows running programs to tolerate failures of their partners with no programmer effort. The prototype implementation of MESNE has a minimum latency of 50 μs, which is comparable to that of GM on the same platforms. A complete implementation is certain to have higher latency, so the approach of providing all protocol computation directly on the Myrinet hardware has been abandoned in favor of using standard protocols as underlying communications layers.

8 Acknowledgements

This matrial is based upon work supported by the National Science Foundation (NSF) under Grants CDA-96-23614 and MIP-9420352 and by the Army Research Office (ARO) under Grant DAAG559710076. Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the ARO.

The authors would also like to thank Thomas V. Thompson, Lisa Durbeck, and John Hollerbach for their valuable contributions to this work.

References

[CD97] Cezary Dubnicki, Angelos Bilas, Kai Li and Jim F. Philbin "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet" Proceedings of 11th International Parallel Processing Symposium, April 1997

[JC99] Chase, J., Anderson, D., Gallatin, A., Lebeck, A., and Yocum, K. "Network I/O with Trapeze", 1999 Hot Interconnects Symposium. August 1999

[CC92] Cruz-Neira, C., Sandin, D.J., DeFanti, T.A., Kenyon, R., and Hart, J.C. "The CAVE, Audio-Visual Experience Automatic Virtual Environment" Communications of the ACM, June 1992, pp 67-72

[DF96] Filion, D. "Double and Triple-Buffering" http://perplexed.com/GPMega/beginners/dubtrip.htm

[GLP83] G.L. Peterson "Concurrent Reading While Writing" ACM Toplas, 5(1) 56-65 1983

[GB95] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes "Hamlyn: A high-performance network interface with sender-based memory management" In Proceedings of the Hot Interconnects III Symposium, Palo Alto, CA, Aug. 1995.

[GP94] G. Pardo-Castellote, A.S. Schneider "The Network Data Delivery Service: Real-Time Connectivity for Distributed Control Applications," IEEE International Conference on Robotics and Automation, San Diego, CA, 1994

[JMH97] John M. Hollerbach, , Elaine Cohen, William B. Thompson, Rodney Freier, David E. Johnson, Ali Nahvi, Donald D. Nelson, Thomas V. Thompson II, and Jacobsen, S.C., "Haptic interfacing for virtual

prototyping of mechanical CAD designs" ASME Design for Manufacturing Symposium, (Sacramento, CA), Sept. 14-17, 1997. Available at http://www.cs.utah.edu/jmh/ASME97.ps

[JN96] J. Nieplocha, R.J. Harrison, and R.J. Littlefield "Global Arrays: A nonuniform memory access programming model for high-performance computers" The Journal of Supercomputing, 10:197-220, 1996. Available at http://www.emsl.pnl.gov:2080/docs/global/papers.html

[LAG98] "A Software Architecture for Global Address Space Communication on Clusters: Put/Get on Fast Messages." Proceedings of the 7th High Performance Distributed Computing (HPDC7) conference (Chicago, Illinois), July 28-31, 1998. Available at http://www-csag.cs.uiuc.edu/papers/index.html#communication

[LP98] Pyrlli, L. "BIP Messages User Manual for BIP 0.94" LIP/ENS-LYON Technical report TR1997-02, Updated June, 1998. Available at http://www-bip.univ-lyon1.fr/bip.html#doc

[MG97] M. Griebel, G. Zumbusch "Parnass: Porting gigabit-LAN components to a workstation cluster" Proceedings of the 1st Workshop Cluster-Computing, held November 6-7, 1997, in Chemnitz, editor: W. Rehm, Chemnitzer Informatik Berichte, CSR-97-05, pp. 101-124. Available at http://wwwwissrech.iam.uni-bonn.de/research/doc/myri/c/c.html

[MM90] Minsky, M., Ouh-Young, M., Steele, M., Brooks, P.P. Jr., Behensky, M., "Feeling and Seeing: Isues in Force Display," Proc. Symposium on Interactive 3D Graphics, 1990, Showbird, Utah, pp. 235-243

[MRS95] Mark R. Swanson, Leigh B. Stoller "Direct Deposit: A Basic User-Level Protocol for Carpet Clusters" University of Utah Technical Report UUCS-95-003, February 1995. Available at http://www.cs.utah.edu/stoller/pubs/user_man.ps

[ND94] N. Durlach and A. Mavor, eds., "Virtual Reality: Scientific and Technological Challenges." Washington D.C.: National Academy Press, 1994

[NJB95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su "Myrinet, a gigabit-per-second local-area network" IEEE Micro, 15(1):29-36, February 1995. Available at http://www.myri.com/research/publications/Hot.ps

[PH99] Hsieh, P. "Flicker Free Animation" http://www.azillionmonkeys.com/qed/flicker.html

[PTW98] Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. "Modeling of a high speed network to maximize throughput performance: the experience of bip over myrinet." In Parallel and Distributed Processing Techniques and Applications (PDPTA'98), Las Vegas, USA, 1998.

[RAFB98] Bhoedjang, R.A.F., Rühl, T., Bal, H.E., 1998, "User-Level Network Interface Protocols," Computer, November 1998

[SCJ90] S.C. Jacobsen, F.M. Smith, E.K. Iversen, D.K. Backman, 1990, "High Performance, high dexterity, force reflective teleoperator" Proc. 38th conf. Remote Systems Technology, Washington D.C. Nov. pp180-185

[SGI99] GSN White Paper, http://www.sgi.com/peripherals/networking/gsn_whitepaper.html

[SM98] Menke, S., Moir, M., Srikanth Ramamurthy "Synchronization Mechanisms for SCRAMNet+ Systems" Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, June-July 1998

[SP97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors" IEEE Concurrency, vol. 5, no. 2, April-June 1997, pp. 60-73.

[TB98] Bowman, T. "Shared-Memory Computing Architectures for Real-Time Simulation – Simplicity and Elegance", Systran technical paper available from http://www.systrant.com/scramnet.htm, January 1998

[TE95] Thorsten von Eiken, Anindaya Basu, Vineet Buch, and Werner Vogels "U-Net: a User-Level Network Interface for Parallel and Distributed Computing" Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, Dec. 3-6, 1995

[TS92] T. Sheridan, "Telerobotics, Automation, and Human Supervisory Control." Cambridge, MA: MIT Press, 1992

Appendix A The API

This section describes the Prototype MESNE API in detail.

Appendix A.1 Connection establishment and teardown

```
int
mesne_connect_write(int local_unit,
                    int local_port,
                    int reader_id,
                    int reader_port,
                    int number_of_elements,
                    int size_of_each_element);
int
mesne_connect_read(int local_unit,
                    int local_port,
                    int writer_id,
                    int writer_port,
                    int number_of_elements,
                    int size_of_each_element);

int
mesne_disconnect(int cid);
```

The connect calls establish half of a connection, either as reader or writer. The disconnect call tears it down and releases the resources. Because each host can have multiple Myrinet interfaces, the user must specify which one to use (`local_unit`). Each interface is assigned a unique ID , and the user must specify the ID of the remote interface.

To support multiple simultaneous users, each interface supports a small number of **ports**, each of which handles exactly one connection. The user must specify both the local port and the remote port.

We emphasise that this does not follow a client-server model. Either of the partners can connect first, because these calls do not establish a connection between two processes, but rather between two ports. After one process has established the connection, a process on the target machine can join if it calls the opposite connect function and specifies the same two ports with matching element size and count details. Further, after two processes have joined, either one can terminate or call the disconnect function without affecting the behavior of the partner's communications functions. An application which is connected to a partner may also terminate and restart with out affecting the partner.

Upon success, the connect functions return a connection identifier. On failure, it returns -1 and returns an error code in $mesne_errno$. The disconnect function returns 0 on success, -1 if the connection id was invalid.

Appendix A.2 Writing

```
int mesne_begin_write(int cid, int block);

int
mesne_element_write(int cid,
                    int element_number,
                    const void *buffer,
                    int block);

int mesne_end_write(int cid);

void *
mesne_vector_address(int cid);

int
mesne_mark_write(int cid,
                 int element_number,
                 int block);
```

The writer signals that a write of the signal vector is about to begin by calling *mesne_begin_write*

In the prototype implementation, the writer end of the communication is implemented as a simple queue, so it is possible for this to block for network related reasons. The *block* parameter can be set to MESNE_BLOCKING or MESNE_NONBLOCKING. If the first is used, the function will block until the system is ready to allow writes to begin. If the second is chosen, it will return 0 if not yet ready, 1 if ready. It will return -1 on error in both cases.

When the *mesne_begin_write* call has completed successfully, the writer can then perform any number of updates to the signal vector with *mesne_element_write*. The *block* parameter works exactly as before; the call will either block until a write can be performed or return 0 to indicate that the write was not accomplished and 1 to indicate success, depending on the value of *block*.

This function copies N bytes (the size of an element) from the specified buffer into the specified element of the signal vector and initiates a send to the reader. When all partial updates are finished, the writer can publish the newly modified signal vector to the reader by calling *mesne_end_write*. The *mesne_write_element* call must be called only between calls to *mesne_begin_write*, and *mesne_end_write*, or data corruption will occur.

The writer may call *mesne_element_write* for the same element multiple times. The reader's will see the values specified in the last call for that element when *mesne_end_write* is called.

The *mesne_element_write* function introduces a copy (though technically it is a copy from host memory to NIC memory and therefore necessary), so the last two functions are provided to avoid this copy. The first returns a pointer to the N*S byte signal vector which you can access directly. This points directly to the appropriate area of the NIC. You can write to any portion of this signal vector, but afterwards you must indicate to the NIC that you changed the memory directly by calling *mesne_mark_write* for any elements that you changed.

If a reader process terminates and restarts, at the very next *mesne_end_write* call the writer's NIC will automatically update all elements which have ever been written. If a writer process terminates and restarts, the reader's signal vector will not be reset to zeroes. The new writer process can either assume that this data is still valid, or write all of its elements to reset the reader's data.

All of these return -1 on failure, which occurs only if *cid* is invalid, the element number is invalid, or if the connection was not a writer.

Appendix A.3 Reading

```
int
mesne_begin_read(int cid,
                 void (*iterator)(int e,const void *buffer),
                 int block);

int
mesne_element_read(int cid,
                   int element_number,
                   void *buffer);

int
mesne_end_read(int cid);


int
mesne_check_read(int cid,
                 int element_number,
                 void **buffer);

int
mesne_iterate_read(int cid,
                   void iterator(int e, const void *buffer));

int
mesne_set_read(int cid,
               int element_number,
               void *buffer);
```

The *mesne_begin_read* call signifies that the process wishes to examine the most recent consistent signal vector. If `block=MESNE_BLOCKING`, this call will block until the internal data structures have been made consistent, which involves DMA from the NIC and some minor processing, so this blocking is not dependent on any network activity, and will proceed as fast as the NIC can perform the DMA operations. If `block=MESNE_NONBLOCKING`, the fuction will return 0 to indicate that the processing is not yet complete, and 1 to indicate that it is done.

After this call, the signal vector will be consistent with the writer's signal vector with all completed writes applied *as of the time of the last mesne_end_write call.*

The *iterator* parameter will be described after the *mesne_iterate_read* function has been described.

The *mesne_element_read* call will copy the current data stored in the specified element of the signal vector into the specified buffer. It will return a status integer c which describes the data: if (c & $MESNE\_VALID$), then at some point in the history of the communication, this element was set by the writer. Otherwise the data is considered invalid and the element will contain zeroes. If (c & $MESNE\_NEW$), then this element was updated by the writer since the last read. New implies valid. The *mesne_check_read*

function will return this status integer without copying the element data. It will return a pointer to the current data for that element whether or not the data is new or valid.

For a large signal vector, it would be inefficient to call the *mesne_element_read* or the *mesne_check_read* function for every element during each read, since only a few (possibly no) elements may contain new data. Therefore, MESNE provides a mechanism to call a function for only those elements which have new data. The *mesne_iterate_read* function will call the iterator function for each element that has new data, supplying it with the element number and a pointer to the new data. This pointer points to the memory region which the NIC stores partial updates via DMA, so this mechanism also implements zero-copy access to the signal vector. The *mesne_iterate_read* function will return the number of elements in the signal vector which contained new data.

The *mesne_begin_read* call also offers an iteration parameter. Since it is possible for the system to be aware of which elements contain new data and where that data will be located before the DMA operations which actually update that data have completed, the *iterator* parameter of *mesne_begin_read* can be used to process some of this while the NIC is performing the DMA. This routine can be used to update the user's data structures to point to the correct location for each element's data. It is important, though, that the iterator routine supplied does not dereference these pointers, since they are not guaranteed to point to stable data until after the *mesne_begin_read* call has completed.

For example, suppose the signal vector consists of the locations and orientations of 64 objects that are being tracked in real-time by some remote system, and a reader application wants to receive these locations and orientations in order to provide a visual rendering. The program may contain a display list of objects to render, and each entry in this list contains the object descriptions and pointers to the object's location and orientation data. The program can call *mesne_begin_read* and supply a function which simply updates the pointers in the list for a specified element. The *mesne_begin_read* call will call this iterator for any elements which have new data, thus updating the pointers in the display list. When the *mesne_begin_read* call has completed, the data pointed to will be stable, and the rendering can proceed, with no copying of the data.

Note that the pointer to the data for each element does not vary in Version 1, since the signal vector is always reconstructed in the same memory region. So, the iterator will always receive the same pointer for a given element. In Version 2, the signal vector is distributed over 4N element buffers, and a new buffer will be used whenever new data comes in, so the pointer to the data for an element will change. So the buffer pointer argument to the iterator function is really only useful in Version 2.

The *mesne_end_read* function ends the read, and allows the system to begin preparing for the next read. After this call, the internal buffers are being updated for the next read, so the element read, check, and iterate functions must not be called until *mesne_begin_read* is called again.

The integer return value will be -1 on failure, which occurs only if *cid* is invalid, the element number is invalid, or if the connection was not a reader.

Appendix A.4 Detecting the Partner

```
int mesne_status(int cid);
```

Since one requirement stated that the behavior of all the API functions must be transparent to failures of the partner, none of the previously discussed functions can detect whether a partner process is actually participating. A reader with no partner will simply receive no new data, and this is indistinguishable from having a partner that is momentarily not sending data, one that has not joined, one that has disconnected, or one whose host machine has crashed. Similarly, a writer can't tell if the data it is sending is being examined.

So the *mesne_status* call returns an integer code which will be non-zero if a partner is present and participating.

Appendix B The MESNE Algorithm

This section describes in detail the algorithm that lets the NIC distribute incoming packets into a pool of buffers in such a way that it can reconstruct the most recently received consistent signal vector. The algorithms for the two versions differ slightly.

Appendix B.1 Version 1: Buffer Pool in NIC SRAM

For each *mesne_element_write* call, the writer sends one packet containing the new element data and the element number to the receiving NIC. The *mesne_end_write* call sends an EOW packet.

The LANai software running on the receiving NIC maintains the 4N S-byte buffers and several data structures in NIC SRAM. The arrays $In[]$, $Out[]$, $LastReceived[]$, and $CompleteWrite[]$ each contain N two-bit integers (0, 1, 2, or 3). The $In[]$ array specfies, for each element, which buffer will receive an arriving update. If $In[e]$ = i, for example, it specifies that an update for element e will be stored in buffer (iN+e). In the same way, $Out[]$ specifies the buffers forming the most recently received complete signal vector which the reader process is examining. $LastReceived[]$ gives the location of the buffer holding the most recently received data for each element. $CompleteWrite[]$ specifies the buffers which contain the next most recently received complete signal vector which has not yet been examined by the reader.

Sixteen of these two-bit quantities can often be processed together as a 32-bit word. So, the N elements are grouped into N/16 groups. In the following pseudocode, accessing one of these array elements directly returns a two-bit quantity, and accessing a casted version will return 16 two-bit quantities in one 32 bit word. For example, $In[5]$ returns a two-bit quantity which is the in buffer index for element 5. $((32bit*)In[3])$ would return a 32 bit quantity consisting of the in buffer indices for elements 48 through 63.

In order to reconstruct a consistent signal vector efficiently, we must maintain a list of all elements which have been modified in the current incomplete write, and all elements which have been modified in complete writes since the last consistent signal vector was provided to the reader.

$LastReceivedList[]$, $CompleteWriteList[]$, and $OutList[]$ serve this purpose. They list which groups of elements contain at least one modified element. The variables LRLSize, CWLSize, and OLSize contain the sizes of each of these lists. At maximum, each can contain N/16 entries.

The reader process maintains a region of DMA-cabable process memory which is the size of the signal vector. This region provides the reader process with its copy of the signal vector. This copy is guaranteed to be static and contain the signal vector with all completed writes applied, but only between a *mesne_begin_read* call and a *mesne_end_read* call. After a *mesne_end_read* call, the system will update this region to reflect all the writes which had completed at the time of the *mesne_end_read* call. The reader process also maintains some additional DMA-capable host memory to receive two copies of $CompleteWrite[]$ and $CompleteWriteList[]$. (These copies are referred to as Out and OutList in Figure 15.) The list is used to provide the *mesne_iterate_read* functionality, and the two copies of CompleteWrite (one from a previous read, one from the current read) can be compared to quickly determine if an element contains new data. The LANai simply alternates between these two copies for each *mesne_end_read* call.

Now, using these data structures, there are only three events that the NIC must respond to: receiving a data packet, receiving a EOW packet, and receiving a request for a new signal vector from the reader. The first two are network events, and the last is signaled through the reader's communication endpoint; i.e., the *mmap*ed memory shared between the reader process and the LANai.

- **Receiving a data packet for element e:**

```
atomic {
  int inval,outval,cwval, group, slr;

  DMA header of packet from network and determine
  element number e;


  group  = e/16;
  slr    = inval = In[e];
  outval = Out[e];
  cwval  = CompleteWrite[e];

  begin DMAing the rest of the packet (the data)
  to LANai buffer (inval*N + e);

  inval = (inval+1)&3;
  if (inval == outval || inval == cwval) inval = (inval+1)&3;
  if (inval == outval || inval == cwval) inval = (inval+1)&3;

  await DMA completion;

  if (CRC is ok) {

      if (((32bit*)LastReceived)[group]
          == ((32bit*)CompleteWrite)[group]) {
          LastReceivedList[LRLsize++] = group;
      }

      LastReceived[e] = slr;
      In[e] = inval;
  }
  else { /* otherwise, disregard the packet completely. */ }
}
```

- **Receiving EOW:**

```
atomic {

  int loop = 0;
  int group;

  while (loop < LRLsize) {
    group = LastReceivedList[loop++];
    if ( ((32bit*)Out)[group] ==
         ((32bit*)CompleteWrite)[group] ) {
       CompleteWriteList[CWLsize++] = group;
    }

    ((32bit*)CompleteWrite)[group]
       = ((32bit*)LastReceived)[group];
  }

  LRLsize  = 0;

}
```

- **Receiving request for new signal vector from reader:**

```
if (CWLsize == 0) {
   set a status location to indicate
     that no new elements are available.
     DMA this status word to the host;
}
else {

   atomic {
      int loop = 0;

      begin DMAing CompleteWrite and CompleteWriteList to host memory,
         targetting one of the two Out and OutList buffers;
      OLsize = 0;

      /* copy CompleteWrite to Out, and CompleteWriteList to OutList */

      while (loop < CWLsize) {
         group = CompleteWriteWlist[loop++];
         OutList[OLsize++] = group;
         ((32bit*)Out)[group]
             = ((32bit*)CompleteWrite)[group];
      }


      await DMA completion;
   }

   DMA each new element to the reader's copy of the signal
      vector.  OutList contains an entry for each group which
      contained at least one new element. Compare Out
      and CompleteWrite for that group to see which elements
      in the group are really modified, and use Out[e] to
      determine which NIC SRAM buffer to DMA from:
      Buffer number (N*Out[e]+e).

   set a status location to indicate that new elements are
      available, and that the CompleteWriteList DMA'd to
      the host contained CWLsize elements.
DMA this status word to the host.

   CWLsize = 0;

}
```

The atomic sections are needed because packet reception and reader requests are handled by two separate contexts on the LANai, one of which is interruptible, so context switches can occur.

The LANai adjusts the $In[]$ array to provide a new reception area for data packets for each element. When one comes in, it updates $LastReceived[]$ to remember where the most recent data for each element

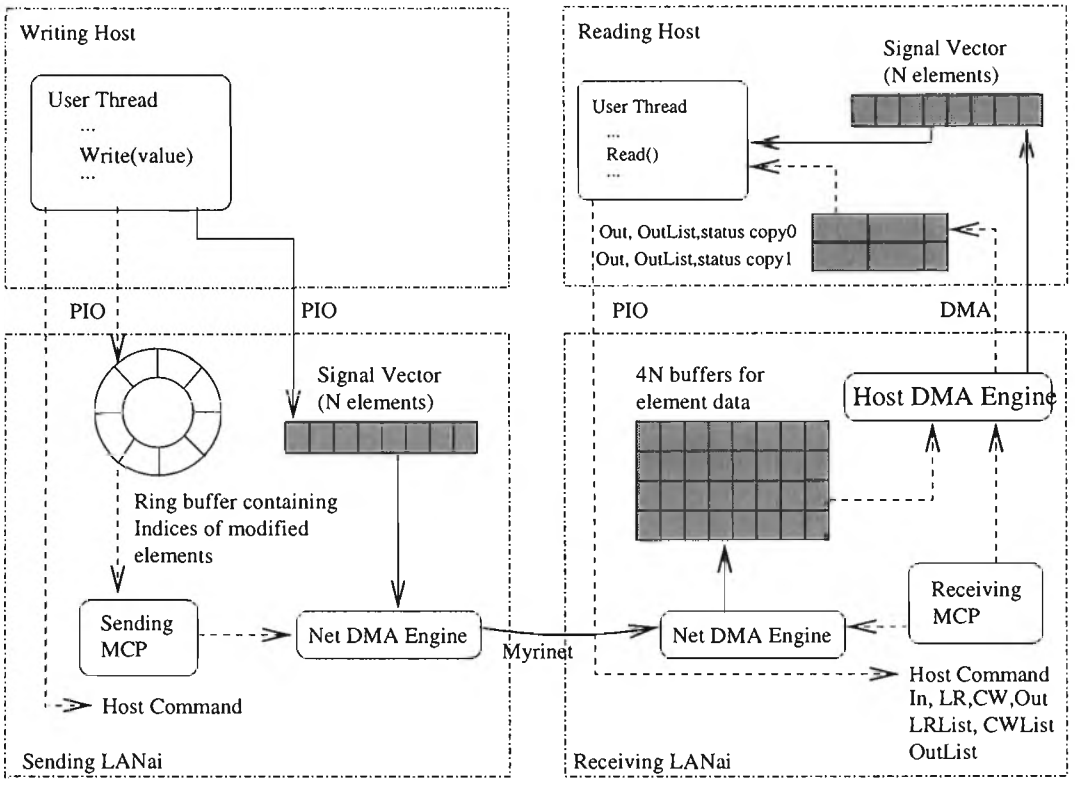


Figure 15: Buffer Placement Schematic for Version 1

has been placed, and updates $LastReceivedList[]$ to include that element's group. When EOW is received, it copies $LastReceived[]$ to $CompleteWrite[]$ for all groups in $LastReceivedList[]$, and adds any new groups to $CompleteWriteList[]$. At this point, $CompleteWrite[]$ contains the indices of all the buffers which make up the most recently received consistent signal vector, and 4 $CompleteWriteList[]$ contains a group entry for each group with at least one new element.

The reader process requests a new signal vector by setting $Host\ Command$. When the LANai reads this command, it clears the $Host\ Command$ location. It then copies $CompleteWrite[]$ to $Out[]$ for all groups in $CompleteWriteList[]$, and it copies both $CompleteWrite[]$ and $CompleteWriteList[]$ to the host $Out[]$ and $OutList[]$ locations by DMA. Then, all new element data is copied to the host by DMA. Finally, the status value is sent to the host by DMA. This status value contains a completion value the size of the $OutList[]$.

The host process then waits for the status location to contain the correct completion value. As soon as it does, the copy of the signal vector in the reader's DMA-capable host memory is a consistent copy of the signal vector with all received complete writes applied. The reader process API code uses the $OutList[]$ to implement $mesne\_iterate\_read$, and compares the current $Out[]$ and previous $Out[]$ to determine which elements have new data. They will differ in any location corresponding to an element with new data. This comparison can be done with a 32-bit exclusive-or.

Appendix B.2 Version 2: Buffer Pool in Host Memory

As in Version 1, the writer simply sends a stream of data packets and EOW packets to the reader. Data packets contain the element number and the data for that element, and are sent by the writer when an element is written. EOW packets are sent when a write is complete (i.e. when the writer calls $mesne\_end\_write$),

and contain no further information.

The LANai software still maintains the In, LastReceived, CompleteWrite, and Out arrays in NIC SRAM, but it keeps the 4N S-byte buffers in the reader process' DMA-capable memory.

$LastReceivedList[]$ and $CompleteWriteList[]$ are still needed, but $OutList[]$ is not.

In addition to the 4N S-byte buffers, the reader process maintains some additional host memory to receive two copies of $CompleteWrite[]$ and $CompleteWriteList[]$, as in Version 1.

Now, using these data structures, there are only three events that the NIC must respond to: processing a data packet, receiving a EOW packet, and receiving a request for a new signal vector from the reader. (In Version 2, all data packets are received into a temporary buffer and their crc's are checked before processing.) The first two are network events, and the last is signaled through the $mmap$ed memory shared between the reader process and the LANai.

- **Processing a data packet for element e:**

```
atomic {

  int inval,outval,cwval, group, slr;

  group  = e/16;
  slr    = inval = In[e];
  outval = Out[e];
  cwval  = CompleteWrite[e];

  begin DMAing this packet's data to host buffer (inval*N + e);

  inval = (inval+1)&3;
  if (inval == outval || inval == cwval) inval = (inval+1)&3;
  if (inval == outval || inval == cwval) inval = (inval+1)&3;

  if (((32bit*)LastReceived)[group] == ((32bit*)CompleteWrite)[group]) {
    LastReceivedList[LRLsize++] = group;
  }

  LastReceived[e] = slr;
  In[e] = inval;
}
```

- **Receiving EOW:**

```
atomic {

  int loop = 0;
  int group;

  while (loop < LRLsize) {
    group = LastReceivedList[loop++];
    if ( ((32bit*)Out)[group] ==
         ((32bit*)CompleteWrite)[group] ) {
       CompleteWriteList[CWLsize++] = group;
    }

    ((32bit*)CompleteWrite)[group]
       = ((32bit*)LastReceived)[group];
  }

  LRLsize  = 0;

}
```

- **Receiving request for new signal vector from reader:**

```
if (CWLsize == 0) {
   set a status location to indicate
     that no new elements are available.
   DMA this status location to the host
     status buffer, targetting one of the two
     copy buffers.

}
else {

   atomic {
      int loop = 0;

      set a status location to indicate
            that new elements are available, and
            that the CompleteWriteList DMA'd to
            the host contained CWLsize elements.

      begin DMAing CompleteWrite, CompleteWriteList,
          and the status location to host memory
          targetting one of the two copy buffers;

      /* copy CompleteWrite to Out */

      while (loop < CWLsize) {
         group = cwlist[loop++];
         ((32bit*)Out)[group]
                = ((32bit*)CompleteWrite)[group];
      }

      await DMA completion;

      CWLsize = 0;
   }
}
```

The atomic sections are needed because packet reception and reader requests are handled by two separate contexts on the LANai, so context switches can occur.

The LANai adjusts the $In[]$ array to provide a new reception area for data packets for each element. When one comes in, it updates $LastReceived[]$ to remember where the most recent data for each element has been placed, and updates $LastReceivedList[]$ to include that element's group. When EOW is received, it copies $LastReceived[]$ to $CompleteWrite[]$ for all groups in $LastReceivedList[]$, and adds any new groups to $CompleteWriteList[]$. At this point, $CompleteWrite[]$ contains the indices of all the buffers which make up the most recently received consistent signal vector, and $CompleteWriteList[]$ contains a group entry for each group with at least one new element.

The reader process requests a new signal vector by setting *Host Command*. When the LANai reads this command, it clears the *Host Command* location. Then it copies $CompleteWrite[]$ to $Out[]$ for all groups in $CompleteWriteList[]$, and it copies $CompleteWrite[]$, $CompleteWriteList[]$, and a status

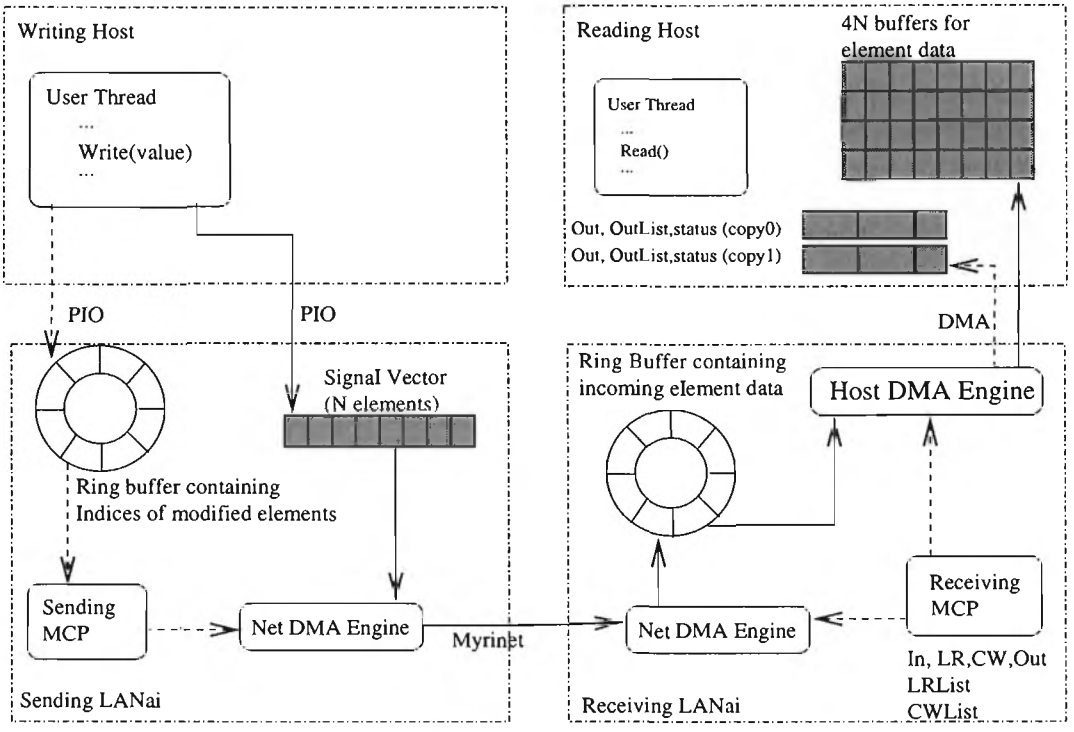


Figure 16: Buffer Placement Schematic for Version 2

value to the host $Out[]$ and $OutList[]$ locations by DMA. The status value contains a completion value and the size of the $OutList[]$.

The host process then waits for the status location to contain the correct completion value. Unlike the previous version, the consistent copy of the signal vector is not assembled as a contiguous signal vector. The method the reader uses to access the consistent signal vector is nonetheless fairly simple. All of the data for the new signal vector is in process memory; it's just distributed over the 4N buffers. $Out[]$ gives the index of the data for any element. I.e., if $Out[e] = i$, for example, the data for element e is in buffer number (iN+e). The reader process will have access to both the current $Out[]$ array and the $Out[]$ from the previous request, so it can determine which elements have new data by looking for indices which have changed. $OutList[]$ allows iterating over the new elements without having to examine them all. An element will be new only if its group appears in this list and the values of the current $Out[e]$ and the previous $Out[e]$ differ.

As soon as it does, all new element data is in host memory. The reader process API code uses the $OutList[]$ to implement $mesne\_iterate\_read$, and compares the current $Out[]$ and previous $Out[]$ to determine which elements have new data. These will differ in any index corresponding to an element with new data. This comparison can be done with a 32-bit exclusive-or.

Appendix B.3 4N vs. 3N buffers

We can now explain the motivation for using 4N buffers rather than 3N. Imagine instead using the simple shared memory algorithm of Figure 8. A packet containing data for element e is received into the buffer with index given by $In[e]$, but note that in this simple algorithm, $In[e]$ is not incremented. When an EOW is received, the $In[]$ array contains the indices of all elements modified in this last complete write. This is exactly the function that $CompleteWrite[]$ serves in the 4N algorithm. So, in the 3N algorithm, we eliminate $CompleteWRite[]$, and when receiving an EOW we copy $In[]$ to $LastReceived[]$.

However, in this simple 3N algorithm, upon an end-of-write, we must increment $In[]$ for all elements

which had new data, wrapping the value and avoiding those already taken by $Out[]$ and $LastReceived|]$. This increases the work required when an EOW is received. This work directly affects the latency, and is not necessarily overlapped with any DMA operations. So, by using 4N buffers, we distribute the work of incrementing $In[]$ over each packet reception and overlap it with DMA operations, and thus trade memory for reduced latency.

There is also a reliability argument for the 4N alorithm, if the multiple buffering is done on the NIC. The 3N algorithm receives data into the $In|]$ location repeatedly until EOW is received. However, if we assume that Myrinet is not totally reliable, we will wish to check the CRC on each received packet. We will not receive the CRC value until the end of the packet. At this point we may have overwritten the valid data at $In[]$ with a data packet that was corrupted and therefore potentially unrelated to this connection at all. The 3N algorithm does not allow us to recover, but the 4N algorithm can recover by simply leaving the values of the data structures $In[]$ and $LastReceived[]$ the same as they were before the packet was received. The buffers with indices specified by $In|]$ therefore become temporary receive buffers whose contents are not accepted until the CRC is checked and the index is moved to $LastReceived[]$. The sliding window system (if one is used) will force a retransmission of such a packet packet eventually if it was just a corrupted but otherwise important packet, and the event will be completely ignored if it was a rogue packet.

Appendix C The Sliding Window Algorithm

We use a typical sliding window algorithm but optimize for the common case of no errors on the network and to tune for the limitations of the LANai.

The sliding window size is fixed at 128, so that sequence numbers (0-255) fit in a byte and are wrapped automatically. This also allows the common operation of taking the sequence number modulus the window size to be converted into an AND operation with 127. This is important since the LANai has no modulus operation in hardware.

Instead of storing a copy of the outgoing packet until the corresponding acknowledgement is received, we use the writer's signal vector directly. The writer performs its partial updates by directly writing into a signal vector kept in NIC SRAM using PIO, as described in Section 5.3.1. The sliding window simply stores the numbers of those elements which have been sent but not acknowledged (and EOF packet indications), thus reducing the memory requred by the sliding window algorithm.

We encounter two potential problems with this approach. First, if the writer writes a given element twice, each write consisting of a PIO write to NIC SRAM followed by queueing the element's number in the send queue, the second PIO write may alter NIC SRAM just as the network send DMA engine is reading that element out of SRAM and sending it into the network, thus corrupting the value which was originally there. However, since the element is queued again, and the reader keeps only the last write, the next send of the element will ensure that this corruption is never perceived by the reader process. Second, assume the writer updates several elements, ends the write, and then begins more updates. If the EOW packet is corrupted or lost, it will have to be resent. However, if those further data packets have been received by the reader before the EOW is resent, the previous write is now corrupted by a new and potentially incomplete write. We cannot wait for the write to complete, since it may not, and we cannot retreive the values of the elements previous to the EOW, since with our scheme the reception may have written over those old values. We prevent this by forcing a writer process to wait for each EOW packet to be acknowledged before writing any further elements.

Rather than starting a timer for each sent packet and retransmitting that packet when the timer expires, the software monitors the age of only the oldest non-acked packet. If it is older than 50ms, it is retransmitted. This may seem large for a low-latency API, but since we expect no errors unless cables are disconnected or switches lose power, this does not affect latency and is optimized for the common case. This single timer scheme is efficiently implemented using the RTC (Real Time Counter) register of the LANai.

The acknowledgements are cumulative, indicating that all packets previous to a given sequence number have been received, and are sent for each EOW, whenever any out-of-window packet is recieved, or when a number of packets equal to half the window size has been received since the last acknowledgement was sent.

Appendix D Memory Protection

As with U-Net [TE95], multiple processes are allowed acces to the network hardware. These processes are protected from interfering with each other by allocating to each a specific region of NIC memory. Memory protection is then obtained by allowing each process to map only the memory allocated to it into process memory using the system call *mmap*.

We require a memory allocation engine which is robust even in the event of process crashes or intentional attempts to interfere with other processes. The close entry point of an IRIX device driver is not guaranteed to be called for each close system call; rather, it is called only when all processes which have opened a device have closed it. If we rely on using the close entry point alone to handle the cleanup of memory allocations, we allow the following scenario: Process A accesses the network normally and begins to use it (i.e. open has been called and resources have been allocated). Then process B tries to connect, allocating NIC memory resources, but is terminated before completing any memory mapping. (The close entry point is not called.) At this point, process A continues to run, but process B's allocated NIC memory resources have not been returned to the system.

To address this problem, we must use the map entry point, since this is the only driver entry point which is guaranteed to be called when a process which has *mmap*ed memory terminates. Here is the protocol for accessing the network. The user process must:

1. *mmap* a page of kernel memory, called the garbage page.
2. call *ioctl* giving all the connect information. The *ioctl* allocates the resources, contacts the partner, and returns a success or failure code to the use. On success, the correct portions of the NIC memory and host memory to *mmap* are also returned.
3. *mmap* the specified areas. Only properly allocated areas can be successfully *mmap*ed.
4. *unmap* the garbage page.

After the first step, any further failure of the user process will result in call to the unmap entry point of the driver, allowing it to clean up any allocations that may have been made in the second step. Once the third step is complete, other mappings exist, and so the garbage page map is unnecessary. A process which does not follow the first and second steps will not be allowed to allocate resources. After the last step, the device driver can properly clean up all resources, including potentially active network connetions, whenever the process terminates or crashes.

In order to simplify the code, the device driver single threads all accesses to the driver routines. Thus, only one process at a time can request connection resources. Since these routines are only called for connection establishment and teardown and not for communication, this is not a bottleneck.

Appendix E Connection Establishment

The device driver and the LANai software work together to establish and tear down connections between ports. The LANai software maintains the signal of each port, and the signal of the relationship with the partner (potentially) connected to that port. LANai software consists of a writer manager, a reader manager, a connection manager, and an interrupt manager.

The LANai software responds to these commands from the device driver and to network events through the connection manager, the writer manager, and the reader manager, as illustrated in figures 17, 17, and 17. Handling of data packets and EOW packets is done by the interrupt manager. If the temporary reception buffers are placed in host memory, the interrupt manager also handles to-host DMA; otherwise this is handled by the reader manager.

The valid port signals and partner signals referred to in the manager figures are described in tables 2 and 3. Here, EOF stands for end-of-frame, which means either end-of-read or end-of-write, depending on context.

Port States	
Name	Meaning
UNUSED	Port not currently in use by a host process
ACQUIRED	Device driver is attempting to forge a connection using this port
RELEASED	Device driver has attempting to tear down a connection using this port
IN_USE	Port is currently in use by a host process

Table 2: The valid port states

Partner States	
Name	Meaning
UNCONNECTED	No communications partner on connected port
PRECONNECT_AWAITING_EOF	Partner detected, awaiting first EOF
PRECONNECT_PROCESSING_EOF	Partner detected, processing first EOF
CONNECTED	Parner connected
RELEASING	Local port is trying to disconnect, notifying managers
RELEASED	Managers notified, notifying partners

Table 3: The valid partner states

The driver can send three commands to the LANai software: ACQUIRE, RELEASE, and ABORT When establishing a connection, the driver will repeatedly give the acquire command until the port state becomes IN_USE, UNUSED, or too many attempts have been made. When attempting to tear down a connection, the driver will repeatedly give the RELEASE command until the port state becomes UNUSED, or until too many attempts have been made. The driver uses the ABORT command to clean up in case either operation failed after the maximum number of attempts.

As previously explained, the connection manager is designed to allow partners to disconnect or fail without requiring any actions by the host process. Note that, when in the IN_USE state, the connection manager handles connects and disconnects of the partner autonomously. If the writer does not receive an acknowledgement of a data packet within a certain time, it assumes the network or the host has failed and sets the

parter state to UNCONNECTED. When a port is in the IN_USE state and the partner state is UNCONNECTED, the connection manager will periodically send connect packets. When these packets arrive at the partner port (after the network problem is resolved or the host reboots) an acknowledgement will be sent indicating if the partner process is still present. If so, communication will continue. If not, the connection manager will keep probing for a partner process.

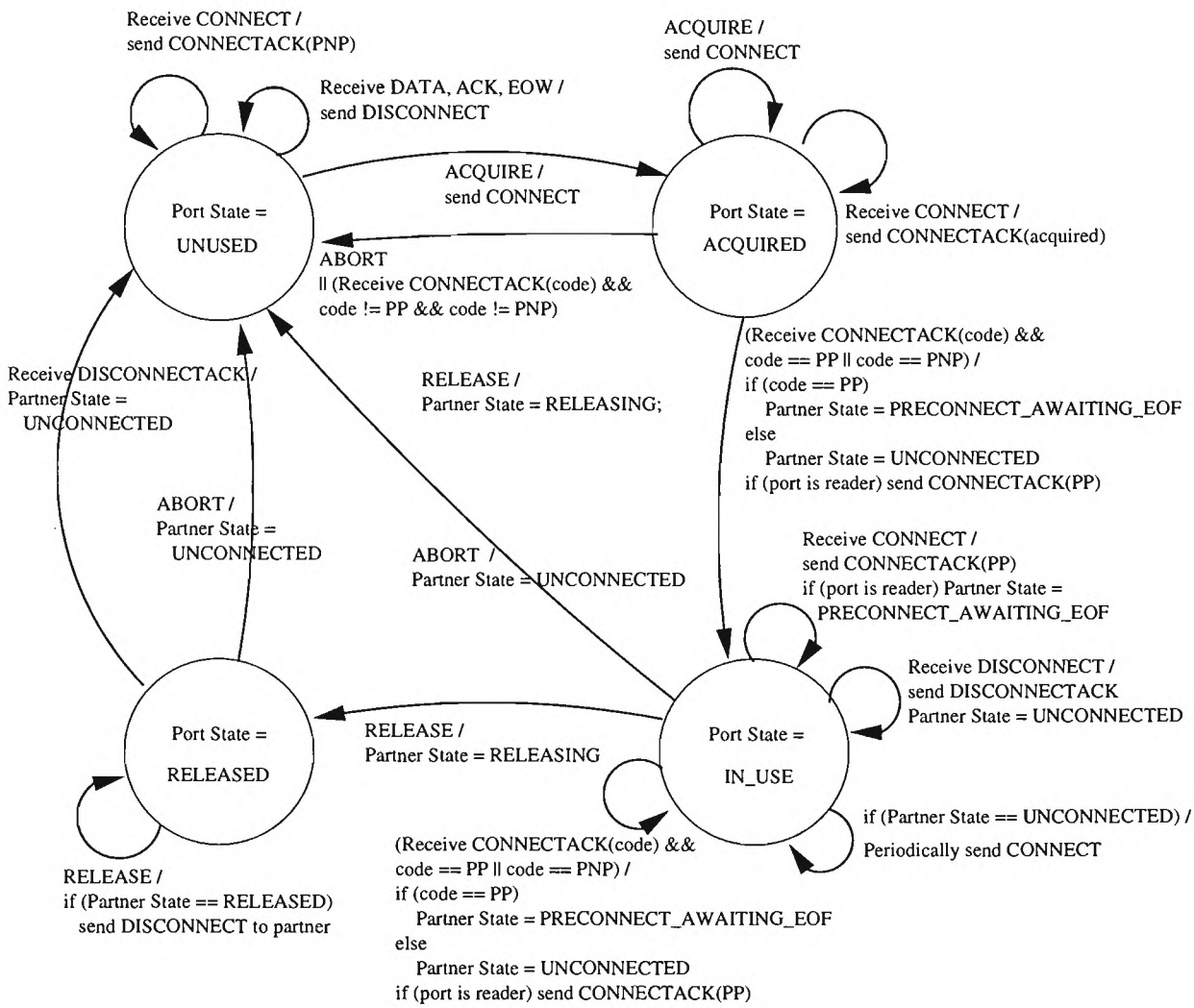


Figure 17: The Connection Manager

The writer manager is responsible for sending a data packet to the partner for every element write performed by the user process. It interacts with the user process through shared memory. It maintains the sending portion of the sliding window, if one is used, and handles retransmissions. The reader manager is responsible for responding to a user's end-of-read command by assembling the consistent signal vector in process memory. Figures 18 and 19 show how the writer manager and the reader manager cooperate with the connection manager to handle the connection state. The actions performed in these figures are explained in detail in Appendix B.

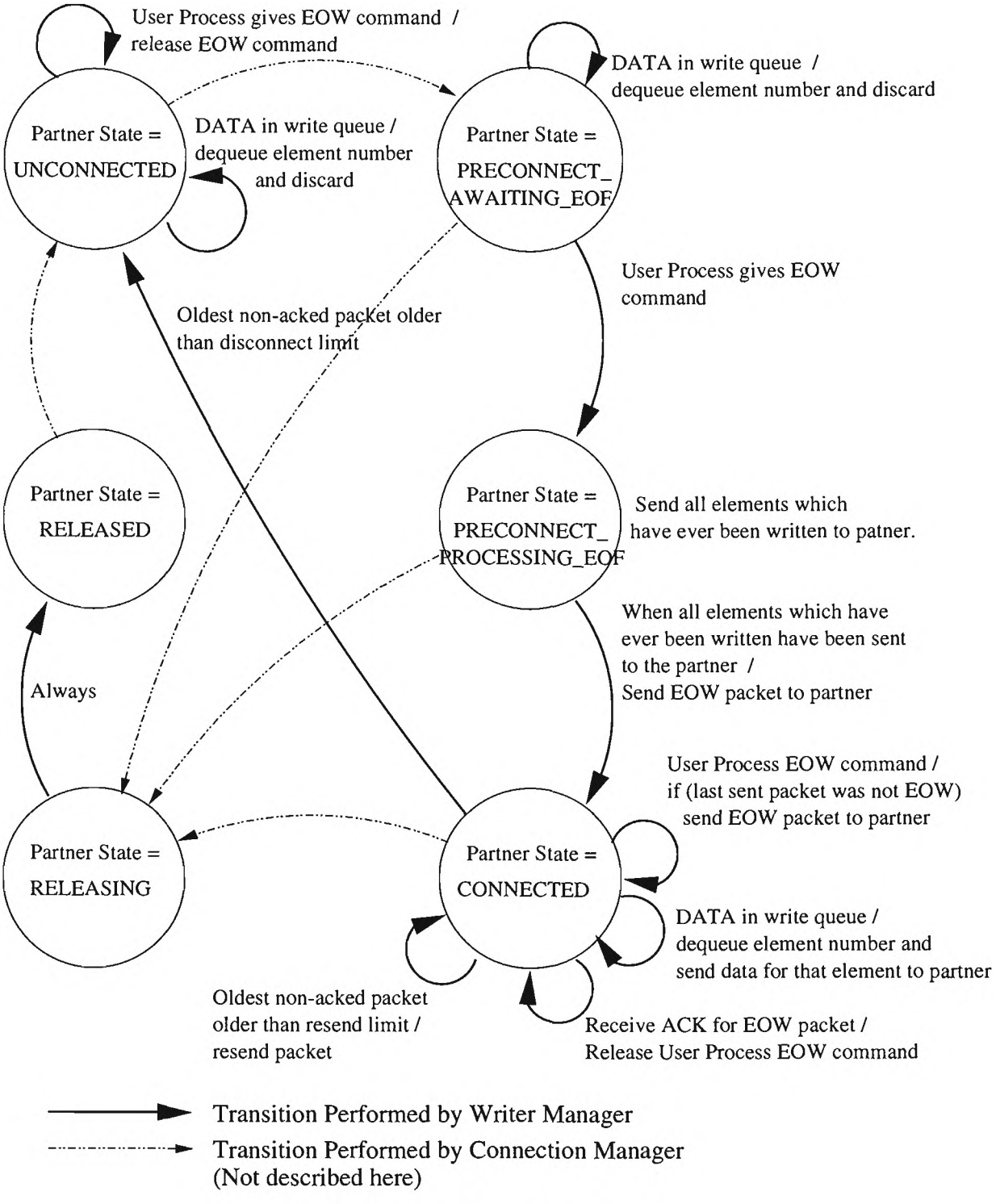


Figure 18: The Writer Manager

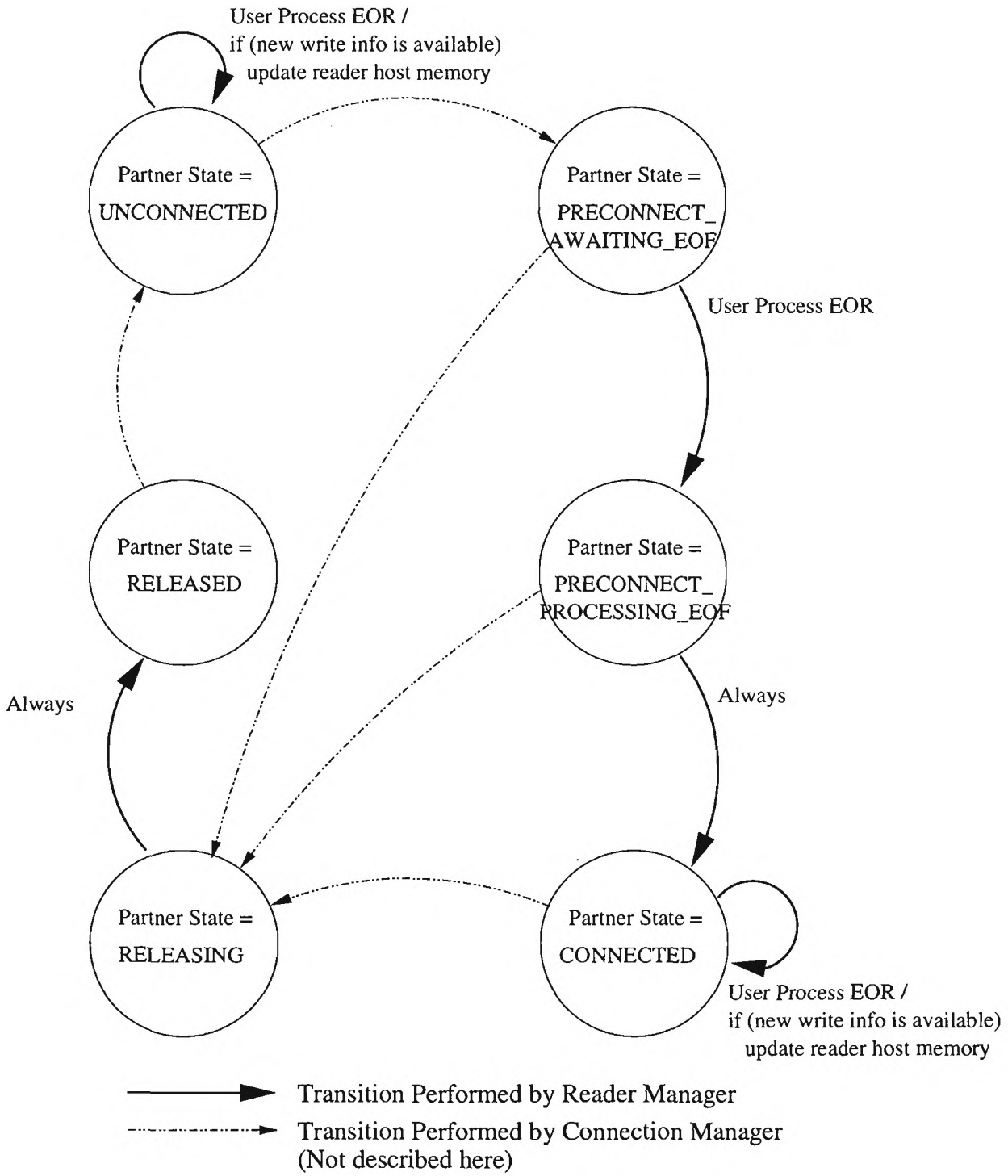


Figure 19: The Reader Manager